®2013, 2014 Eugenia Bahit – www.eugeniabahit.com – Bajo Licencia Creative Commons BY-NC-SA


# Ingeniería de Software y Estructuras de Directorios definitivas en Aplicaciones MVC Modulares

**Lo padecí durante años. Durante años me dediqué a investigar y tratar de descubrir un esqueleto que resultase tan óptimo como irrefutable. La creación de Europio Engine me ayudó mucho en el hallazgo de una respuesta y ahora, tras dos años de constantes pruebas sobre algo más de 70 aplicaciones creadas, puedo contarles mi propuesta sin miedo a cometer errores.**

Enredos, desprolijidades, decisiones «provisorias para siempre», son solo algunas de las tantas causas que generan el malestar y la desesperación que todo programador padece de forma permanente en su carrera como desarrollador de Software y a pesar de haber leído los mejores libros, estudiado con los mejores docentes, seguido los mejores consejos y haberse mantenido actualizado sobre los últimos avances en tecnología, la desgastante sensación nunca desaparece.

Parece una introducción irónica, casi sarcástica, pero no lo es. Es la pura realidad que vivo día a día cuando reviso algún código en GitHub, intento hacer algo de ingeniería inversa sobre alguna app o simplemente escucho padecer a mis alumnos. Pues **la organización del código fuente, la estructura interna de una aplicación y su esqueleto, representan el 80% del éxito de una aplicación**. Y si alguna de estas falla, el fracaso del Software es un hecho inminente.

## La importancia contar con un sistema bien organizado

Las bases de una aplicación modular, no son solo un formalismo técnico de moda. Por consiguiente, sus principios de **portabilidad, encapsulación e independencia**, deben ser respetados sin margen de error.

Un sistema modular es aquel que se compone de un número desconocido de aplicaciones denominadas «módulos» y que permite la incorporación o eliminación de un módulo sin alterar en absoluto las aplicaciones

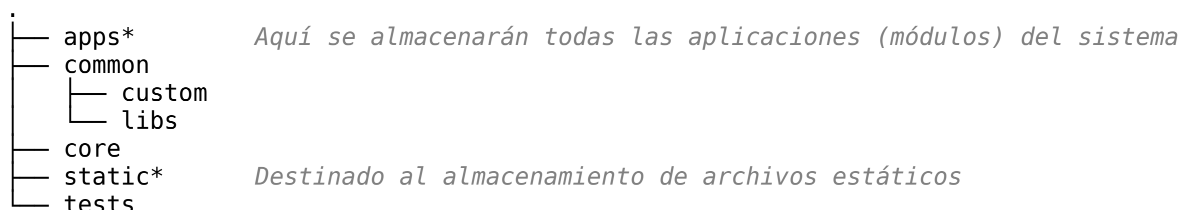
restantes. De esta forma, un sistema modular que hoy cuenta con un módulo de ventas y otro de costos, mañana podrá contar con un módulo extra de seguridad social sin que los módulos de ventas y costos se vean afectados.
Por otra parte, es norma inviolable, que un sistema modular cuente con un núcleo independiente, que pueda mantenerse, actualizarse y modificarse sin que ello afecte a los módulos del sistema.

Por supuesto que todo esto no se logra solo con una estructura interna coherente. Pero igual de cierto es, que contar con un esqueleto irrefutable, nos soluciona el 80% del problema.

## Propuesta estructural

Como comenté al principio, tras un largo período de pruebas e investigaciones, logré dar con una estructura que al momento, a demostrado dar los mejores resultados. Esta estructura que propongo no es nueva. Parte de ella puede verse sugerida por grandes autores como **Martin Fowler**[4] y aplicada por numerosos *frameworks* en diferentes lenguajes. Yo simplemente he tomado esta arquitectura y he creado un híbrido que toma lo mejor de ésta sumando unos toques propios muy sutiles.

Exteriormente, el **esqueleto estructural de un sistema modular MVC** (independientemente del lenguaje con el que se encuentre creado), debería verse exactamente como el siguiente (algunos nombres pueden variar):




Los asteriscos * indican que el nombre del directorio puede variar

### Sobre los nombres variables

El directorio **apps** tiene por objetivo el almacenamiento de los módulos del sistema (más adelante veremos su estructura interna). Por lo tanto, nombres como `applications` y `modules` también son válidos. Es importante que se conserve un nombre el plural, puesto que la estructura de directorios de un sistema, es también una forma de documentación. Los nombres en inglés representan una sugerencia universal que puede ser bien entendida por cualquier programador/a independientemente de su nacionalidad o procedencia.

El directorio **`static`** debe ser una carpeta fácilmente trasladable (*portable*) cuyo único objetivo sea el almacenamiento de archivos estáticos inherentes a la GUI del sistema. Por lo tanto, admitirá nombres tales como `media`, `site_media`, `static_server`, etc. dependiendo del tratamiento que dicho directorio fuese a tener.

4 http://www.martinfowler.com

```
static
├── css
├── html
├── img
├── js
├── sb-admin-v2
└── synthaxhl
```

En el ejemplo anterior, los directorios `sb-admin-v2` y `synthaxhl` corresponden a librerías de terceros y por lo tanto, su estructura interna ha sido conservada.

## Sobre la finalidad de los directorios

La finalidad de las carpetas `apps` y `static` fue explicada anteriormente. Con respecto a las restantes, nos encontramos con los directorios `common` y `core`, los principales del sistemas y por ello es muy importante entender la diferencia entre ambos.

El directorio **`core`** tiene como única finalidad el almacenamiento de archivos y librerías del núcleo de la aplicación. Son archivos del núcleo, todos aquellos que resulten **indispensables para el arranque y funcionamiento del sistema** y que a la vez **puedan ser portados a otro sistema sin requerir modificaciones**. Si un archivo del `core` requiriese un mínimo cambio para servir a otro aplicación, sería un error que podría deberse a:

- La necesidad de generar una directiva de configuración variable;
- La necesidad de mover el archivo fuera del núcleo;

El directorio **`common`** suele ser similar al directorio `core` en el sentido de que todo el sistema necesita de él para funcionar. Sin embargo, a pesar de su naturaleza esencial, todos los archivos son comunes al sistema en el que se encuentran pero su portabilidad hacia otro sistema sería imposible. Es decir, que se trata de **archivos y librerías comunes al sistema que los contiene pero no a otro**, incluso aunque la similitud entre dos o más sistemas se vea afectada por la casualidad de dicha similitud, a reutilizar archivos de un `common`.

Finalmente nos encontramos con la carpeta **`tests`**, claramente destinada a las pruebas unitarias del sistema. Es sumamente importante que aunque el equipo de desarrollo aún no haya logrado implementar técnicas de programación avanzadas como TDD o pruebas unitarias, incluya esta carpeta en los sistemas que diseñe, pues servirá en un futuro al tiempo que sirve como forma de respuesta rápida a si se cuenta o no con *tests*. Al ver esa carpeta vacía, con una sola vista rápida, el programador podrá saber que el sistema no cuenta con pruebas unitarias y que no ha sido desarrollado mediante TDD.

## Directorios por dentro

La estructura interna de cada directorio, varía de acuerdo al mismo y a factores inherentes al proyecto. Sin embargo, ciertas normas y sub-carpetas deberán ser respetadas como se describe a continuación. Comenzaré

®2013, 2014 Eugenia Bahit – www.eugeniabahit.com – Bajo Licencia Creative Commons BY-NC-SA

por lo más simple e iré hacia lo más complejo.

Lo más aconsejable para el directorio **`tests`** es que éste replique la misma estructura de directorios que la del SUT. Esto permitirá que al momento de empaquetar el sistema para su distribución, puedan suprimirse los *tests* sin más esfuerzo que eliminar el directorio correspondiente.

La estructura interna del directorio **`static`** debe indefectiblemente quedar a criterio de los diseñadores gráficos y *maquetadores*. Si no se contase con la experiencia de profesionales gráficos en el proyecto, puede emplearse una subestructura tradicional (que no por ser tradicional será la más óptima) que garantice un orden mínimo y al mismo tiempo, facilite su comprensión. Tradicionalmente, una estructura interna óptima, sería aquella que basase su organización en el tipo de archivos. Para ello, se podría contar con una carpeta por tipo de archivos estáticos: `html`, `css`, `js`, `img`, `audio`, `video`, `pdf`, etcétera.

Cuando se trabajase con plantillas o esqueletos gráficos prediseñados como *Frameworks* o soluciones tales como *Bootstrap*, será aconsejable colocar dentro del directorio estático la carpeta original de dicha librería, respetando su estructura interna intacta ya que esto simplificará el proceso de futuras actualizaciones.

La estructura interna del directorio **`core`**, claramente depende de forma directa, del núcleo que se utilice y de cómo éste haya sido creado. Si se tiene previsto desarrollar un núcleo propio, lo más aconsejable sería organizar los subdirectorios por responsabilidades y/o tipo de herramienta. Esto es equivalente a modularizar el núcleo. Algunos ejemplos serían: *orm*, *sesiones*, *helpers*, *interfaces*, etc.

El directorio **`common`**, sin dudas, es el que mayor tendencia a la desorganización genera y sin embargo, es el que más necesita de una rigurosa disciplina en su estructura. Como les digo a mis alumnos, *«la carpeta common debe organizarse con un régimen militar»*.

La carpeta `common` suele poder resolver sus necesidades con 2 subdirectorios: `custom` y `libs` y jamás debería tener archivos en la raíz, salvo contadas excepciones bien justificadas. A lo sumo, la estructura interna más compleja que pueda verse en la raíz, podría verse como la siguiente:

```
common/
├── core
├── custom
├── libs
├── tools
└── plugins
```

En esta estructura, cada subdirectorio cumpliría una función específica:

- **`common/core:`** respetando la estructura interna del *core* del sistema, su responsabilidad sería la de almacenar cualquier extensión del núcleo. Ejemplos de ellos podrían ser clases heredadas del objeto de algún ORM o -más común- herencia de helpers.

- **`common/custom`:** esta es la carpeta más importante y paradójicamente la menos utilizada. La tendencia universal del programador, es descargar librerías o herramientas de terceros y confundir la libertad que el Software Libre nos da con una práctica de programación. Que un Software sea libre y permita modificar el código no implica que exista una buena práctica de programación que diga «modifica los archivos originales y arruina la posibilidad de mantenerlos actualizados». Los archivos de terceros JAMÁS deben modificarse. Hacerlo obliga a mantener la misma versión del software, por el resto de los días sin posibilidad de instalar actualizaciones, con todos los riesgos de seguridad que ello implica. El software de tercero siempre queda intacto y es en esta carpeta *«custom»* que haremos los cambios necesarios. ¿Cómo? Sobreescribiendo lo menos posible aquellas clases o funciones que se deseen personalizar.

  Una buena técnica y muy práctica consiste en crear una `CustomSomeClass` que herede de una `SomeClass` existente y sobrescriba el método que se quiere modificar, invocando previamente (siempre que sea necesario) al método `parent` correspondiente. Cuando se tratase de una función, simplemente se crearía una `CustonSomeFunction` para una `SomeFunction`.

- **`common/libs`:** son todas las librerías de terceros.

- **`common/tools`:** son todos aquellos guiones (*scripts*) propios creados con finalidades puntuales, como pueden ser generadores de *backups*, *cleaners*, funciones para envío de correo electrónico, *scripts* ejecutados por el `cron`, etc.

- **`common/plugins`:** los complementos suelen ser un tema muy discutido y el hacerlo tiene argumentaciones válidas desde ambas partes. Un complemento es una aplicación completa (como un módulo del sistema) pero que en vez de poder ser utilizada por un usuario es utilizada por el programador en el desarrollo de los módulos del sistema. De esta forma, un generador de formularios accesible por el navegador, que permita al usuario crear un formulario de contacto para el *frontend* de su sistema, sería un módulo mientras que una clase generadora de formularios que el programador utilizaría en la lógica de una vista para crear el formulario que el usuario utilice para generar los suyos, sería un plugin.

  Por lo tanto, un *plugin* podría considerarse complemento del *core* (es aquí donde se discute) y la línea que marca la diferencia es muy delgada y está dada por la respuesta a la pregunta *«¿viviría una aplicación sin este complemento?»*. Un ejemplo claro está en *Europio Engine*. Al ser *Europio* un núcleo propiamente dicho, el generador de formularios se considera un *plugin* ya que se podría crear cualquier tipo de aplicación y/o sistema prescindiendo de éste. Sin embargo, si se creara un *framework* utilizando *Europio Engine*, el mismo generador de formularios que hoy se considera un *plugin*, debería formar parte de las herramientas del núcleo del *framework*.

## Estructura interna de un módulo

Esta estructura fue la que más me costó hallar y más años me llevó tanto investigar como probar y evaluar resultados.

Cuando se trata de una arquitectura MVC, claramente cada módulo tendrá sus directorios `models`, `views` y `controllers`, pero ahí, no puede ni debe concluir el esqueleto.

Un módulo es un Software completo. Llamémosle micro-aplicación si se lo desea, pero no deja de ser un software que emplea un núcleo común. Como tal, tendrá sus propios requerimientos y necesidades y por lo tanto, **no podrá prescindir de los siguientes archivos y directorios**:

```
app-name/
├── models/
├── views/
│   └── templates/
├── controllers/
├── helpers/
├── config.ini
├── config.ini.dist
├── app-name.sql
├── license
└── README
```

La estructura anterior es la única que ha demostrado hasta el momento, verdadera capacidad de portabilidad.

Entre los directorios, nos podremos encontrar con los siguientes destinos:

**`models/`**
almacenaje de todos los modelos de la aplicación (módulo del sistema)
**`views/`**
almacenaje de la parte lógica de todas las vistas del módulo
**`views/templates/`**
en caso de que el módulo posea templates HTML personaizados (que deban ser leídos pero no servidos) es aconsejable almacenarlos en este directorio, facilitando así la portabilidad del módulo.
**`Controllers/`**
almacenaje de los controladores del módulo
**`helpers/`**
almacenaje de clases y/o funciones ayudantes propias de este módulo

Los archivos de la raíz del módulo, es aconsejable que sean solo y únicamente los propuestos, considerando que cada uno contará con la siguiente responsabilidad:

**`config.ini / config.ini.dist`**
Todo aquel dato que pueda ser variable dependiendo del contexto, debe *setearse* en una variable global o preferentemente, en una constante. Utilizar un archivo de configuración por secciones, es una excelente alternativa ya que la mayoría de los lenguajes, posee funciones que permiten analizar este tipo de archivos (parsing) y utilizar su información. Un ejemplo de este archivo DEBE distribuirse (portarse) en un archivo de distribución .dist

**app-name.sql**

El módulo necesitará sus propias tablas en el sistema. Este archivo debe contener todas y cada una de las instrucciones SQL necesarias para que con tan solo importar el archivo a una base de datos, se ejecuten las consultas requeridas para que el módulo funcione sin rodeos. Una buena práctica es ir completando este archivo en tiempo real a medida que el desarrollo demande cambios en la DB.

**README**

De verdad, este es el archivo más importante. Puede ser un simple archivo de texto con instrucciones cortas. Cada vez que el desarrollo demande un cambio de configuración en la plataforma, la instalación de un software, librería o herramienta de terceros, dicha información se agregará en tiempo real en este archivo, bajo el título «requerimientos e instalación»

**license**

Un simple archivo de texto con el nombre de la licencia bajo la cual se distribuye el módulo y preferentemente un enlace hacia el texto de la licencia, será la mejor forma de portar y distribuir micro aplicaciones.

Tu saldo de PayPal
cóbralo desde cualquier parte del mundo
Clic aquí
Tarjeta de débito prepaga MasterCard
Compras con tu tarjeta alrededor del mundo
Extracción de dinero en efectivo desde Cajeros Automáticos
Cuenta bancaria virtualen USA
(para transferir el dinero desde PayPal)
Payoneer
MasterCard
Regístrate ahora y recibe USD 25.- de regalo
con tu primera carga de USD 100.-