

Cube

Configuració de l'Apache i el PHP

Hem de tenir instal·lat l'Apache 2.2 (instal·lat a *c:/Apache2.2*) amb ssl (si volem treballar amb https) al nostre servidor amb el PHP 5.2.4 (instal·lat a *c:/Apache2.2/ php5.2.6*) com a mínim (testejat i funcionant amb 5.2.6 i 5.3). Els exemples i les configuracions són sota windows però són vàlids tant per linux com macos.

El fitxer httpd.conf ha de semblar-se a això:

```
# Activem el PHP
LoadModule php5_module "c:/Apache2.2/php5.2.6/php5apache2_2.dll"
PHPIniDir "C:/Apache2.2/php5.2.6"
AddType application/x-httpd-php .php

# Activem els hosts virtuals
LoadModule vhost_alias_module modules/mod_vhost_alias.so

# Activem https
LoadModule ssl_module modules/mod_ssl.so

# Activem el mod rewrite
LoadModule rewrite_module modules/mod_rewrite.so
```

La configuració del virtual host (extra/httpd-vhosts.conf) serà:

```
# Activar virtual host
NameVirtualHost *:80

# Configurar cube (projectes)
<VirtualHost *:80>
    ServerName projectes
    DocumentRoot "C:/projectes/web"
    DirectoryIndex index.php
    RewriteEngine On
    <Directory "C:/projectes/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

Afegir una entrada al hosts (C:\Windows\System32\drivers\etc\hosts) i comprovar amb un ping que ens connectem en local.

```
127.0.0.1    projectes
```

El directori del php ha d'estar a les variables d'entorn per a poder executar scripts php des de la consola de cube (C:\Apache2.2\php5.2.6). Les extensions actives del php han de ser:

```
extension=php_curl.dll
extension=php_gd2.dll
extension=php_gettext.dll
extension=php_ldap.dll           # obligatòria per treballar amb directori actiu
extension=php_mbstring.dll       # obligatòria per treballar amb caràcters
extension=php_mysql.dll
extension=php_mysqli.dll
extension=php_oci8.dll
extension=php_pdo.dll            # obligatòria per treballar amb pdo
extension=php_pdo_mysql.dll
extension=php_pdo_oci.dll
extension=php_pdo_sqlite.dll
extension=php_soap.dll           # obligatòria per treballar amb webservices
extension=php_sockets.dll
extension=php_sqlite.dll
```

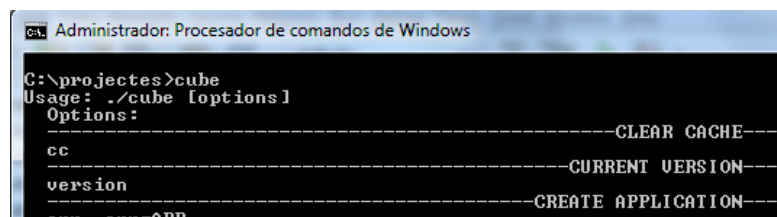
Reiniciar l'Apache per a que els canvis tinguin efecte.

Instal·lació de Cube

Obtindrem els arxius Font de cube a partir de la versió 1.1 de cube. Els arxius estan separats en dos grups: core (branches/1.1) i plugins (plugins/1.1), perquè el desenvolupament, entenem, que ha de ser separat. Passos:

- **Creem carpeta projecte**, per exemple "projectes" al sistema de fitxers.
- **Baixem engine i web**: Fem un export per si hem de modificar arxius. Svn export <http://portalaplicacions/svn/cube/branches/1.1> C:\projectes\
- **Baixem plugin principal (cubePlugin)**: svn export <http://portalaplicacions/svn/cube/plugins/1.1/cubePlugin> C:\sandbox\plugins\cubePlugin
- **Copiem l'script de consola** (*engine/scripts/cube.bat* per windows, o *engine/scripts/cube* per linux o macos) a l'arrel del projecte (C:\projectes).

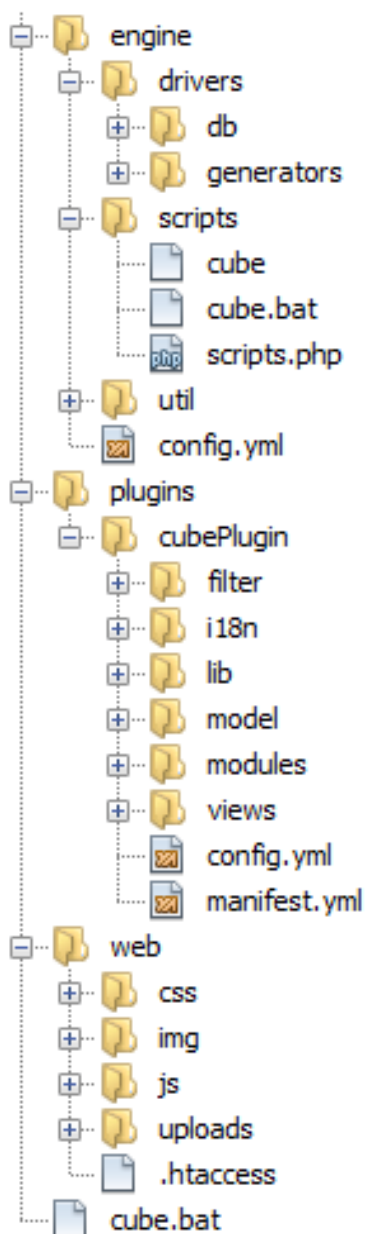
Per verificar que cube s'ha instal·lat bé escriurem en consola "cube" (o "./cube") per veure les opcions que podem fer. Si les visualitzem, cube ha estat instal·lat correctament.



```
C:\projectes>cube
Usage: ./cube [options]
Options:
-----CLEAR CACHE-----
cc
-----CURRENT VERSION-----
version
-----CREATE APPLICATION-----
app -app=APP
```

Si volem subversionar el nostre projecte, ho podrem fer sense problemes perquè hem fet export de tot el codi. Si volem tenir les actualitzacions al dia haurèm de fer checkout d'aquestes carpetes (engine, web, plugins) a banda i reescriure les que ja tenim al nostre svn, i mirar que els merges que es facin no tinguin conflictes.

Sistema de fitxers resultant



Engine

Arxius del nucli de cube. Conté:

- **Drivers:** Conté drivers per accés a base de dades(mysql, oci, ldap) i generadors de codi (només es fa servir cubeGenerator) que serveixen per generar els administradors. (parlarem més endavant)
- **Scripts:** conté els scripts de consola.
- **Util:** Llibreries de cube.
- **Config.yml:** Configuració principal. Els seus valors poden ser reescrits tant a l'aplicació com als mòduls.
- **Plugins.yml:** configuració dels plugins instal·lats a cube (només lectura). Es genera quan s'afegeixen plugins.

Plugins

Carpeta amb afegits de cube. A una instal·lació simple només cal que contingui *cubePlugin*, que té informació de:

- **Filter:** filtres a executar: execucions internes.
- **I18n:** internacionalitzacions bàsiques
- **Views:** vistes disponibles (default, log, print). La vista log es fa servir a la consola web de depuració.
- **Models:** models essencials:
 - **mailer:** enviament de mail
 - **rss:** creació de rss
 - **myuser:** usuari de sessió
 - **util:** utilitats (treball amb arrays, manipulació amb ip's, validació nif, generació mdb, zip, ...)
- **Modules:** mòduls per defecte:
 - **default:** planes de error, seguretat
 - **plugins:** administrador de plugins
 - **validation:** validació de l'usuari
- **Config.yml:** Configuració pròpia del plugin.
- **Manifest.yml:** Arxiu manifest del plugin. Sense aquest el plugin no es visible pel framework.

Web

Conté informació pública (accessible per web):

- **Css, Img, Js i uploads:** fulls d'estil i icones de la consola web de depuració, imatges per defecte, javascripts necessaris per executar cube (jquery i inicialitzacions de components). La carpeta uploads està preparada per la pujada de fitxers, amb seguretat apache. Es fa servir al component input/file.
- **.htaccess:** configuració apache per reescriure (mod_rewrite) les url, planes d'error i seguretat de les carpetes.

Cube.bat

Arxiu (còpia de scripts/cube.bat) per fer servir la consola cube (versió Windows)

Hello World

Primers conceptes

Per veure per pantalla un simple Hello World hem de comprendre primer el sistema que fa servir cube, que és el mateix que symfony 1.X i està compost de aplicació – mòdul – acció.

L'aplicació conté un controlador al qual podem configurar 3 paràmetres:

- Aplicació, dins el directori d'aplicacions (/apps)
- Cache, que crea arxius temporals de configuració
- Debug, que mostra la consola web de debug, molt útil en desenvolupament

L'aplicació conté configuració de les rutes que es faran servir per cridar als mòduls i passar els paràmetres corresponents. També conté altre informació com els plugins activats, internacionalització (i18n, per defecte activada) i altres.

El mòdul es comporta com a paquet d'accions a realitzar per l'aplicació i facilita l'ordenació de codi, i conté arxius de configuració (que poden modificar els valors d'aplicació) i vistes necessàries per recolzar, si fa falta, a l'acció.

L'acció conté el codi a realitzar. Aquesta acció pot donar com a resultat un text simple (sense embolcall) o un complex (amb template i/o layout). Si no es defineix res a l'acció sempre tenim un template que porta el mateix nom que l'acció, però aquest comportament es pot desactivar.

Creació de l'aplicació – mòdul - acció

Una vegada tenim clar aquests conceptes passem a fer la nostra primera aplicació. :

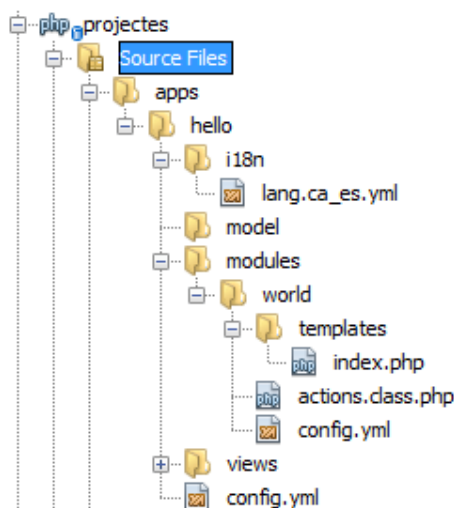
- **Creació de l'aplicació:** executem a consola “cube app –app=hello”
 - Això crea a /web dos controladors, un per producció (hello.php) i un altre per desenvolupament (hello_dev.php).
 - També crea /apps/hello i una sèrie de subcarpetes on trobem:
 - I18n: configuració de llenguatge (per defecte ca_es)
 - Model: conté la informació del models de l'aplicació (si en té)
 - Modules: conté els mòduls de l'aplicació (està buit)
 - Views: si necessitem crear vistes específiques per l'aplicació
 - Config.yml: arxiu de configuració
 - Encara no podem executar res a l'explorador perquè no hem creat cap acció!
- **Creació del mòdul:** executem a consola “cube module –mod=world –app=hello”
 - Crea /apps/hello/modules/world que conté:
 - Templates: vistes per les accions (per defecte conté el template index)
 - Actions.class.php: accions per aquest mòdul
 - Config.yml: configuració específica del mòdul
- Anar a la configuració de i18n i canviar la línia 3 per “ module: Hello” (amb els 4 espais al davant!!)
- **Esborrar cache:** executem a consola “cube cc”. Això prepara cube per crear els arxius necessaris de configuració per a la nova aplicació.

- **Verificar** mitjançant la url: <http://projectes/hello.php/world/index>
 - Es pot comprobar fàcilment els components de la ruta: hello.php (controlador de producció de l'aplicació hello), world (mòdul de l'aplicació hello), index (acció a realitzar).
 - De fet també seria vàlid (1) <http://projectes/hello.php/world> i (2) <http://projectes/hello.php>. Ara explicarem perquè.
 - Veiem que tenim el text decorat amb un template i un layout que podriem estalviar. Després veurem com no decorar el resultat.

La plana web hauria de tenir quelcom semblant a:



El sistema complet d'arxius de l'aplicació que de la següent manera:



Routing

El sistema de routing de cube també és semblant al de symfony 1.X i consisteix a definir una línia per ruta i varis paràmetres que podem modificar-la i afegir-ne de nous. També hi ha una sèrie d'opcions per afinar més la ruta amb expressions regulars. Aquesta configuració la podem trobar a `/apps/hello/config.yml` a l'etiqueta `routing`. En el nostre cas:


```
homepage:
  url: /
  param:
    module: default
    action: index
default_index:
  url:('/:module')
  param:
    action: index
default:
  url:('/:module/:action/*')
```

Podem veure que tenim definides tres rutes (`homepage`, `default_index` i `default`). La primera defineix la ruta si posem a la url <http://projectes/hello.php>. Executa l'acció `index` del mòdul `default`. Com és possible que funcioni si no existeix el mòdul `default`? Hi ha una petita trampa i és que **cube**, per defecte, afegeix a totes les aplicacions el mòdul `default` (mitjançant el plugin `cubePlugin`) que és imprescindible per fer servir la lògica de les planes no trobades, planes sense permís, planes per defecte i altres.

Com ja podreu imaginar l'etiqueta `url` permet configurar la url de la ruta en qüestió i l'etiqueta `param` que serveix per modificar o afegir paràmetres de la url. Hi ha una altra etiqueta que porta per nom `requirements` que serveix per afinar el tipus de paràmetre definit a `param`.

Així tenim que la ruta `default_index` s'activa si posem un mòdul qualsevol de l'aplicació (nosaltres només tenim un i es diu `world`) i l'acció a realitzar és `index`. De forma anàloga la ruta `default` s'activa quan la url conté **mòdul/acció/qualsevol cosa** (fixeu-vos que té una `/*`, encara tenir una `/` s'activarà aquesta ruta si posem tant sols **mòdul/acció** sense la `/` final).

Podem fer aquestes comprovacions veient el resultat a la consola web de debug. Haurem d'executar llavors el controlador de desenvolupament de l'aplicació `hello` (`hello_dev.php`) per exemple amb la url `http://projectes/hello_dev.php`.

Apareixerà un quadre a la part superior - esquerra amb les lletres LOG. Farem clic per veure tota la informació que s'ha realitzat per executar l'acció. Filtrem la informació que ens interessa, que és la de la ruta, fent clic a la icona .

La informació és la següent:

LOG				
CUBE V.1.1 - Debug Console - Dev Mode				
Main (341,948 Ms)				
TIME	METHOD	TYPE	MESSAGE	INFO
283,736 ms	ROUTE::PARSEROUTE		Activate routing 'homepage' : '/' for route '/'	✓
283,818 ms	ROUTE::PARSEROUTE		Routing vars: array ('app' => 'hello', 'file' => '/hello_dev.php', 'module' => 'default', 'action' => 'index', 'view' => 'index',) Request vars: array ('PHPSESSID' => '067424305d9724f384c1a702fc55bfc0',)	✓
293,467 ms	CONTROLLER::INIT		Enabled modules configuration: default	✓

Podem veure que s'activa la ruta *homepage*, que els paràmetres de la ruta són **app=hello**, **file=/hello_dev.php**, **module=default**, **action=index** i **view=index**. Aquestes 5 paraules en negreta estan reservades i es poden modificar a la ruta per “ocultar” accions. Per últim podem veure que s'ha activat la configuració del mòdul default (que està a cubePlugin/modules)

Si ara canviem la ruta, per exemple a http://projectes/hello_dev.php/world

TIME	METHOD	TYPE	MESSAGE	INFO
281,411 ms	ROUTE::PARSEROUTE		Activate routing 'default_index' : '/:module/' for route '/world'	✓
281,491 ms	ROUTE::PARSEROUTE		Routing vars: array ('app' => 'hello', 'file' => '/hello_dev.php', 'action' => 'index', 'module' => 'world', 'view' => 'index',) Request vars: array ('PHPSESSID' => '067424305d9724f384c1a702fc55bfc0',)	✓
286,137 ms	CONTROLLER::INIT		Enabled modules configuration: default	✓

Veiem que la ruta activa és *default_index*, i que ara **module=world**. Això és perquè s'ha assignat a la variable *:module* del routing el valor *world*, que és el que conté la URL.

Si provem amb la url http://projectes/hello_dev.php/world/index tenim que:

TIME	METHOD	TYPE	MESSAGE	INFO
261,780 ms	ROUTE::PARSEROUTE		Activate routing 'default' : '/:module/action/' for route '/world/index'	✓
261,858 ms	ROUTE::PARSEROUTE		Routing vars: array ('app' => 'hello', 'file' => '/hello_dev.php', 'module' => 'world', 'action' => 'index', 'view' => 'index',) Request vars: array ('PHPSESSID' => '067424305d9724f384c1a702fc55bfc0',)	✓
266,385 ms	CONTROLLER::INIT		Enabled modules configuration: default	✓

Les variables són les mateixes però la ruta que s'activa és *default*.

Com podem afegir altres paràmetres?

Podem afegir altres paràmetres que no estan a la ruta:

- directament a la url. Per exemple: http://projectes/hello_dev.php/world?parametre=1

En aquest cas les variables d'enviament (*Requests vars*) tindrien **parametre=1**

281,596 ms	ROUTE::PARSEROUTE		Routing vars: array ('app' => 'hello', 'file' => '/hello_dev.php', 'action' => 'index', 'module' => 'world', 'view' => 'index',) Request vars: array ('parametre' => '1', 'PHPSESSID' => '067424305d9724f384c1a702fc55bfc0',)	
------------	-------------------	--	--	--

- Modificant el routing. Per exemple al *default_index*, afegint a param una entrada *parametre: 1*.

266,775 ms	ROUTE::PARSEROUTE		Activate routing 'default_index' : '/:module/' for route '/world'	✓
266,855 ms	ROUTE::PARSEROUTE		Routing vars: array ('app' => 'hello', 'file' => '/hello_dev.php', 'action' => 'index', 'module' => 'world', 'view' => 'index',) Request vars: array ('PHPSESSID' => '067424305d9724f384c1a702fc55bfc0', 'parametre' => '1',)	✓

La nova configuració del routing és aquesta. Això és útil si volem passar paràmetres extra a la nostra ruta. Cal tenir en compte, però a l'hora de recuperar els paràmetres a l'acció, saber d'on surt aquest paràmetre extra.

```
routing:
  homepage:
    url: /
    param:
      module: default
      action: index
  default_index:
    url: '/:module'
    param:
      action: index
      parametre: 1
  default:
    url: '/:module/:action/*'
```

I18n

Com és que modificant un arxiu de configuració, sense modificar la vista, ha aparegut el text “Hello World”?

Si heu estat atents al codi generat veureu que tant a la configuració de l'aplicació com a la del mòdul (*config.yml*) tenim a l'apartat **settings** - **i18n**, el **default_lang=ca_es** i **enabled=true**. Això vol dir que aquest mòdul té activada l'opció d'internacionalització (i18n pels amics), i que mitjançant l'arxiu *i18n/lang.ca_es.yml* hem canviat el valor de l'etiqueta *default:module* a “Hello”.

Si seguim investigant trobarem que a l'acció del mòdul **world** es fa una crida al mètode `_echo` de la classe `Viewer` (`Viewer::_echo('default:module').' World'`). Aquest mètode llegeix les etiquetes del llenguatge activat actual (en el nostre cas `ca_es`). El valor “Hello” està a l'arbre d'etiquetes **default – module** i això ho denotem amb **'default:module'**. Després concatenem amb la paraula “World” que màgicament ens ha sortit al codi. Bé, màgicament no, realment és el nom capitalitzat del mòdul.

Si us heu fixat els arxius de configuració tenen l'extensió *.yml*. Són arxius YAML que també es fan servir al symfony 1.X i són molt útils perquè l'estructuració és molt fàcil d'entendre per un usuari i facilita així la cerca de qualsevol dada de configuració.

Yaml

Hi ha molts documents explicant les característiques d'aquest tipus d'arxiu però bàsicament:

- Notació **etiqueta: valor**, o **etiqueta: "valor"**, o **etiqueta: 'valor'**
- **Etiqueta** és un objecte si va entre {}. *Etiqueta: {subetiqueta: valor}*
- Es pot escriure també amb una tabulació de 2 espais:

```
etiqueta:
  subetiqueta: valor
```

- **valor** és un array si va entre []. *Etiqueta: [valor1, valor2]*
- podem escriure **valor** també amb notació de subetiqueta:

```
etiqueta:
  - valor1
  - valor2
```

- **on/off** és equivalent a **true/false** i són booleans.
- **#** comenta tota una línia, encara que estigui al mig de la mateixa. Per exemple, no tindria cap efecte la línia: **Parametre: 1 # aquest és el primer parametre**
- No cal posar cometes als valors encara que distingeix segons quins tipus de dades i escapa caràcters de forma automàtica.

Acció

Que és una acció realment? Una acció és un mètode d'una classe.

```
<?php
class WorldActions extends Actions
{
    public function executeIndex($request)
    {
        $this->content=Viewer::_echo('default:module').' World';
    }
}
?>
```

- A l'exemple que ens ocupa veiem que tenim la classe *WorldActions* que identifica les accions del mòdul *world*. Ha d'extendre forçosament a la classe *Actions* perquè és la que té tots els mètodes per treballar amb accions (redirecció, establir template i layout, mode debug, ...)
- Cada acció del mòdul és un mètode *public* amb el prefix *execute* i l'acció capitalitzada. Així tenim que hem definit l'acció *index* per al mòdul *world*.
- Sempre es passa un objecte *\$request* que **no** coincideix amb *\$_REQUEST* de PHP perquè té informació addicional. Per extreure una variable d'aquest objecte podem fer-ho de 3 formes totalment equivalents:
 - `$par=$request->parametre1;`
 - `$par=$request->getParameter('parametre1','valorSiNoExisteix');`
 - `$params=$request->get(); $par=$params['parametre1'];`

La primera forma és la més ràpida, però sempre dona *null* si aquell paràmetre no existeix. Per això podem fer servir la segona que assegura un valor en cas de que el paràmetre no es passi per *request*. Si volem agafar més d'un valor, potser sigui

interessant fer servir la tercera forma que retorna un *array* de valors, molt més ràpid que fer crides al mateix objecte per tots els paràmetres.

- Automàticament l'acció es decora amb la vista *Index* perquè és el nom de l'acció sense el prefix *execute*. Si volem cridar una altra vista haurem de fer servir el mètode *setTemplate*. Per exemple `$this->setTemplate('principal')`.
- Les variables locals de l'acció que porten *\$this* són visualitzades per la vista i traduïdes amb el nom de variable sense *\$this*. Per exemple `$this->data` és converteix en *\$data* a la vista. És una forma còmoda de passar paràmetres.

Vista

Cube està dissenyat amb un patró MVC. Fins ara hem vist només la part de controlador i ara veurem en què consisteix la part de la vista. Cube 1.1 té dos sistemes de vistes:

- **Sistema de templates:** està fortament relacionat amb les accions i és l'encarregat de decorar l'acció. Normalment porta el mateix nom que l'acció però podem fer que una vista concreta decori una acció. Per passar variables de l'acció a la vista només hem de crear una variable local de l'acció (que és una classe) i automàticament es converteix en una variable a la vista. Si ens fixem en la variable *content*:

Actions.class.php

```
<?php
class WorldActions extends Actions
{
    public function executeIndex($request)
    {
        $this->content=Viewer::_echo('default:module').' World';
    }
}
```

Templates/index.php

```
<?php echo $content; ?>
```

- **Sistema de components:** és l'encarregat de pintar a pantalla qualsevol objecte més o menys complex, per exemple, una caixa de text, un grid, un formulari, un visualitzador d'imatges, un cercador, ...
 - El plugin de cube (cubePlugin) ja porta una sèrie de components per poder treballar però podem estendre'ls, reescriure'ls i crear-ne de nous.
 - Es criden amb el mètode *view* de la classe *Viewer* amb 3 paràmetres:
 - El component a visualitzar (*input/text*)
 - Un array de paràmetres (*array('internalname'=>'parametre1')*)
 - Tipus de vista (per defecte, *default*)
 - Els components estan definits a les carpetes *views* de qualsevol aplicació, mòdul, model i plugin. Normalment hi ha dos definits dos tipus de vista, *default* i *print*, i corresponen a la vista per defecte i la vista per impressió, però en podem fer d'altres, segons les necessitats.
 - La ruta del component (primer paràmetre de *Viewer::view*) coincideix amb la ruta fins els sistema de fitxers a partir de *views/default* (tercer paràmetre de *Viewer::view*) separat per / sense espais.

- Si cridem a una vista que no existeix no apareixerà res per pantalla i no donarà cap error, simplement no farà res. Això és molt útil si no volem visualitzar segons quin component a un tipus de vista.

Si heu observat als fitxers de configuració (config.yml) sota l'etiqueta de view trobem una sèrie de valors:

```
view:
  http metas:
    content-type: 'text/html; charset=UTF-8'
  metas:
    title: hello
    description: 'cube project'
    keywords: 'cube, project'
    language: ca
    robots: 'index, follow'
  has_layout: true
  layout: default
```

- **http_metas i metas** fan referència als metas que escriurem a la capçalera de la plana. Podem modificar la codificació i el títol entre d'altres.
- **Has_layout i layout** són els encarregats de decorar mitjançant un layout el diferents templates que pugui tenir el mòdul (si la configuració que estem tractant es la del mòdul) o tots els templates de l'aplicació. Si posem *has_layout* a **false** (també off) a la configuració del mòdul *world*, veurem per pantalla que l'acció no es decora. Si el deixem a true (també on) l'acció es decora amb el layout *default* que es troba , per defecte, al plugin de cube (cubePlugin) a la ruta */plugins/cubePlugin/views/default/layouts/default.php*.

Quina diferència hi ha entre el layout default i el tipus de vista default?

Internament, cube treballa amb el tipus de vista (o visualitzador) *default*, per tant, tots els components que fa servir, com per exemple els layouts, estan sota la carpeta default. Si volem canviar el tipus de vista ho podem fer passant *viewer* com a paràmetre a la url. Per exemple:

http://projectes/hello_dev.php/world?viewer=print

Veiem, ara que l'acció es decora d'una altra manera. De fet el que hem fet es presentar la mateixa informació sense capçalera ni peu de plana i afegir un text informatiu de la data i l'hora de la impressió. Això ho hem pogut fer ràpidament perquè al visualitzador print no tenim definits totes les vistes de *page_elements* (header.php) i hem modificat d'altres (footer.php).

Aquest sistema permet crear vistes específiques sense canviar el contingut principal de la plana, per exemple, per crear la versió pdf o exportar a excel.

```

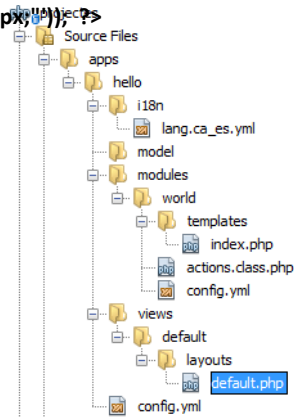
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<?php $module=Controller::getRoute('module'); ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Layout <?php echo $module; ?></title>
<?php echo Viewer::includeMetaJsCss(); ?>
</head>
<body>
<div style="margin: 30px;">
<h2>Layout especial pel mòdul <?php echo $module; ?></h2>
<?php echo Viewer::view('input/longtext', array(
    'value'=>$vars['body'],
    'js'=>'style="width:200px;height:100px;"'), ?>
</div>
</body>
</html>

```

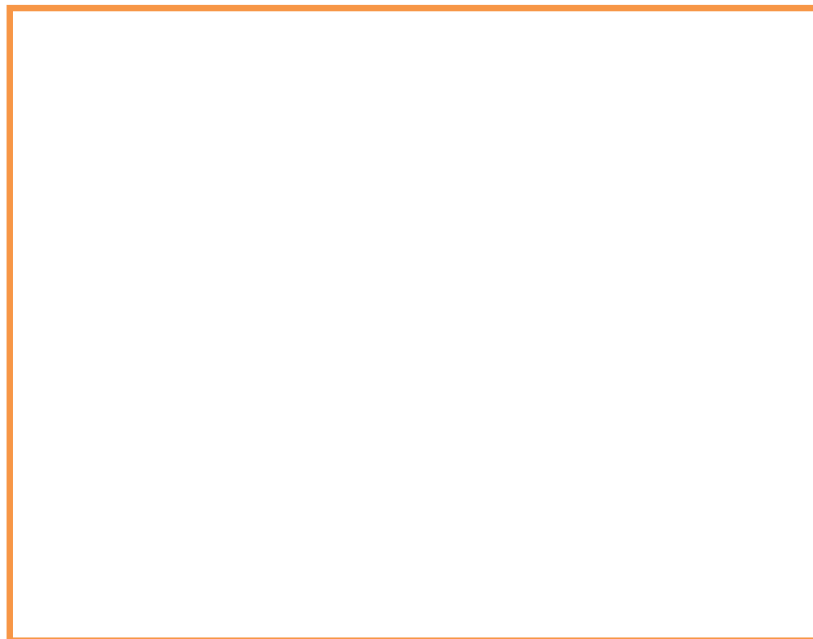
El layout default pot ser reescrit dins de l'aplicació o dins del mòdul si creem dins de la carpeta views la mateixa estructura d'arbre. Provarem a crear un layout default, específic per al mòdul world de l'aplicació hello:

- Copiem */plugins/cubePlugin/views/default/layouts/default.php* a */apps/hello/modules/world/views/default/layouts/default.php*

per a deixar tot el sistema d'arxius d'aquesta manera:



- Copiem el codi següent al nou *default.php* de mòdul:



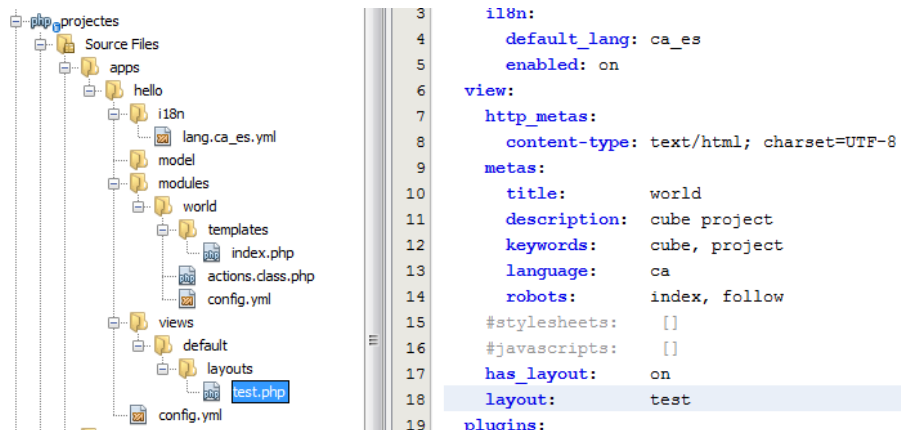
- *Controller* és la classe del controlador principal. Conté, entre d'altres el mètode **getRoute** que ens dona informació de la ruta que s'ha executat. Aquesta informació coincideix amb les variables que vam veure a l'apartat de routing. Podem extreure aquests valors independentment (p.ex només el mòdul) com es mostra al codi.
- *Viewer* és la classe de la vista. El mètode **includeMetaJsCss()** inclou a la plana tota la informació que hem configurat als config.yml i per codi (si ha fet falta). El mètode **view** és el creador i renderitzador de components. Amb

aquest codi crearem un text llarg (representat per un textarea) amb un valor i estil específics.

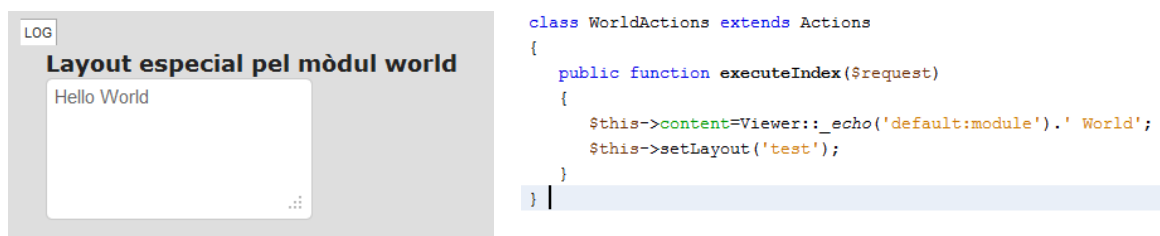
- `$vars` és un array que conté les variables passades al component `layout`. Per exemple, dins el component `input/longtext` que té dos paràmetres (`value` i `js`) tindrem un array `$vars['value']` i `$vars['js']`. Hi ha paràmetres comuns a tots els components i altres específics.

- Esborrarem la cache perquè hem introduït nous fitxers al sistema i actualitzem la plana del navegador.

Que ha passat? O més ben dit, que no ha passat? Esperàvem que el layout `default` s'actualitzés però no ha sigut així. Això és perquè el la prioritat de les vistes té en compte els plugins actius i com sempre està actiu `cubePlugin`, totes les seves vistes són prioritàries. Hauríem de crear un altre plugin amb més prioritat que `cubePlugin`. Això ho farem més tard. Ara, per veure com queda el nostre layout nou caldrà renombrar-lo a, per exemple, `test` i modificar la configuració per a que executi el layout `test`.

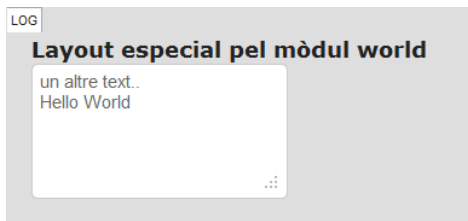


Fixeu-vos que hem renombrat a `views/default/layouts/default.php` per `test.php` i hem canviat al `config.yml` del mòdul l'etiqueta `layout` amb el valor `test`. També podem fer aquest canvi, per codi, sense haver de modificar l'arxiu de configuració, escrivint a l'acció `$this->setLayout('test');`. Ara ja podem actualitzar la plana web:



Veiem que el layout ha canviat completament. No és molt normal posar un “textarea” que decori el template, però això és un exemple.

Si afegim al template un text de prova veiem que el layout queda igual i ara el valor del textarea és el nou contingut del template.



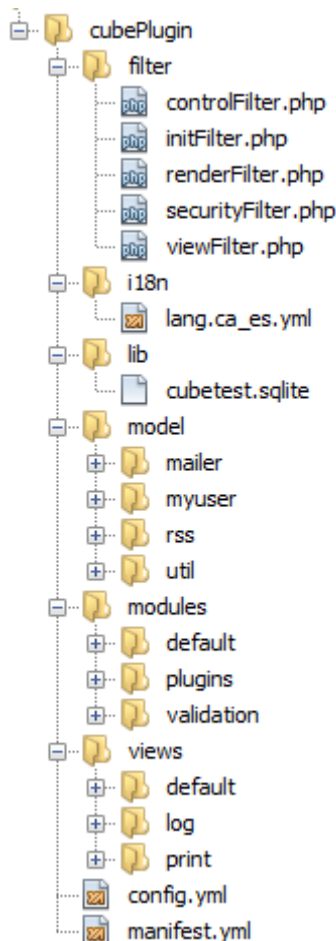
Codi de l'arxiu templates/index.php

```
1 un altre text..
2 <?php echo $content; ?>
```

Esta clar que si desactivem el template, també desactivem el layout. Amb el codi `$this->setTemplate(false)` no s'executa la part de la vista de l'acció.

Plugins

Els plugins, en essència, són paquets que tenen una sèrie d'utilitats que poden servir en més d'una aplicació. De fet presenten la mateixa estructura que una aplicació a excepció de que tenen el sufix Plugin, tenen un arxiu manifest.yml i la configuració del plugin és accessible posant el nom del plugin davant (es comentarà més tard).



A l'instal·lació només hem afegit cubePlugin, que es de vital importància per a que funcioni el framework. A Cube 1.1 es va optar per posar tota la informació de filtres, i18n, models, mòduls i vistes al plugin. Aquest plugin ha de ser el de més baixa prioritat per a poder estendre o reescriure qualsevol servei que conté.

La carpeta filter conté els filtres de execució interns del framework. Es comentaran al següent apartat.

Podem crear qualsevol carpeta a banda de les reservades pel framework (i18n, model, modules, views). Per exemple, tenim una carpeta lib on tenim una base de dades en sqlite per fer tests, que a hores d'ara està incompleta, a falta de fer un mòdul de test que treballi amb aquest arxiu.

L'arxiu manifest.yml és obligatori a l'hora de publicar un plugin a cube. És un arxiu que conté informació del plugin (autor, descripció, versió, ...). Sense aquest arxiu cube no pot detectar la creació d'un nou plugin i per tant és com si estigues donant de baixa. A l'esborrar cache, cube torna a cercar els plugins actius per tal d'actualitzar la configuració general.

Inclusió d'un Plugin a una Aplicació

Per tal d'utilitzar un plugin (vistes, i18n, model) a una aplicació cal modificar l'etiqueta `plugins` de la configuració de l'aplicació, o del mòdul. Per defecte, al crear la nova aplicació s'inclou `cubePlugin` a la configuració de l'aplicació (i es deixa en blanc a la del mòdul):

```

2   plugins:
3     - cubePlugin

```

També podem haver posat la configuració en una línia: *plugins: [cubePlugin]*

Aquesta inclusió permet tenir accés a totes les vistes del plugin, tenir configurada la internacionalització i poder fer servir tots els models que porta, però no permet accedir als seus mòduls. Això és perquè els mòduls s’han d’incloure a demanda dins de l’aplicació i només tenen efecte a la configuració de l’aplicació (Seria incoherent definir l’accés a un mòdul quan ja estem dins un altre mòdul).

Plugin activat i plugin inclós

Un plugin pot estar inclós a una aplicació però aquest pot estar desactivat a tot el framework. Per incloure un plugin ho farem com hem explicat anteriorment. L’aplicació detectarà que aquest plugin està permès dins l’aplicació i el farà servir.

Si un plugin, previament inclós a una aplicació, es desactiva, aquella aplicació no pot fer servir el plugin. Això és així per a no haver de tocar les configuracions de les aplicacions i desactivar-lo (en comptes de treure’l de la configuració a totes les aplicacions) temporalment, per exemple, perquè ja no s’utilitza o per modificació del codi d’algunes de les seves parts que entren en conflicte amb altres plugins.

Per activar o desactivar un plugin ho farem mitjançant el mòdul *admin* que porta *cubePlugin*. Però com hem vist afegint el plugin a l’etiqueta *plugins* no podem accedir als mòduls. Cal fer-ho llavors dins l’etiqueta *enabled_modules* sota l’etiqueta *settings*:

```

settings:
  i18n:
    default_lang: ca_es
    enabled: true
  enabled_modules:
    - default

```

Settings conté informació del llenguatge que es farà servir a l’aplicació (i18n) i dels mòduls de plugins als que pot accedir (*enabled_modules*). No és obligatori que un plugin estigui inclós a l’etiqueta *plugins*, però probablement necessitarà serveis del plugin i per tant s’haurà d’incloure forzoçament.

Quan creem una nova aplicació sempre afegim el mòdul *default* que conté les accions per defecte com són els errors 404, 403, 500, planes de seguretat, planes en construcció, index de planes, ...

Per tant, si volem veure l’administrador de plugins haurem d’activar-lo a la nostra aplicació afegint-lo al llistat de mòduls disponibles. Si no l’afegim i escrivim al navegador la url <http://projectes/hello.php/plugins> possiblement sortirà una plana com la següent:



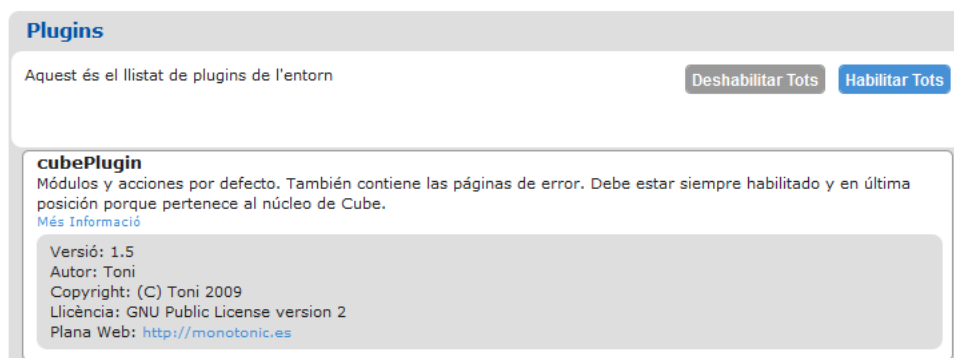
És clar, la plana no existeix. El sistema espera un mòdul plugins a l'aplicació hello, però l'aplicació no en té cap. El mòdul que hem d'activar es diu *plugins*:

```
enabled_modules:
```

```
- default
```

```
+ plugins
```

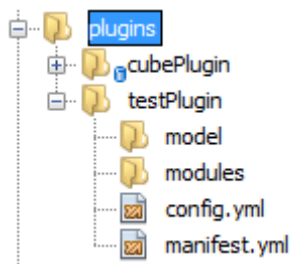
Ara, actualitzem la plana i veurem l'administració de plugins. Només hauria d'aparèixer el cubePlugin actiu. Si surten més plugins és perquè hi ha una configuració antiga instal·lada. El que hem de fer es esborrar del sistema d'arxius engine/plugins.yml i esborrar cache (per eliminar la còpia guardada a cache d'aquest mateix arxiu).



Aquest mòdul s'executa des de cubePlugin però podem crear el nostre mòdul propi default al nostre plugin i com tindrà una prioritat més alta que cubePlugin

Creació d'un plugin

Per crear un nou plugin ho podem fer des de la consola. Per exemple volem crear un plugin amb el nom test, per fer proves. Escriurem: `cube plugin --plugin=test[Plugin]`. De forma anàloga a la creació d'una aplicació, cube crearà una carpeta amb el nom del nom plugin.

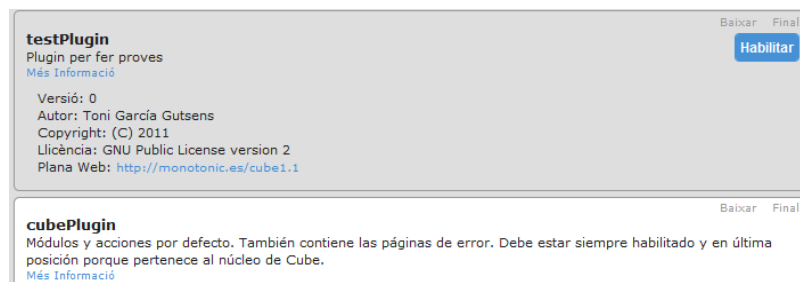


```
author:
version: 0.0
description:
website:
copyright: (C) 2011
license: GNU Public License version 2
cube_version: 2011081801
```

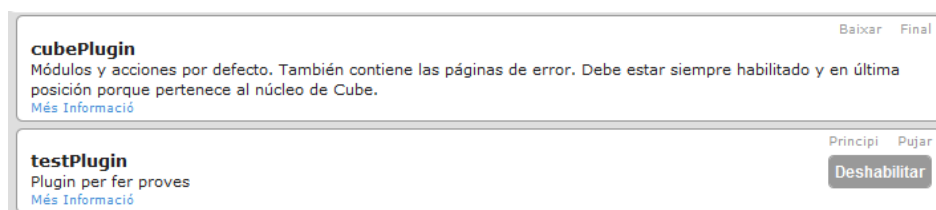
Veiem que ens crea una carpeta per afegir models i una altra per afegir mòduls així com l'arxiu de configuració i l'arxiu de manifest. Canviem la configuració del manifest segons amb les dades de l'autor, versió, descripció, plana web, copyright, llicència i versió compatible de cube:

```
author: Toni García Gutsens
version: 0.0
description: Plugin per fer proves
website: http://monotonic.es/cube1.1
copyright: (C) 2011
license: GNU Public License version 2
cube_version: 2011081801
```

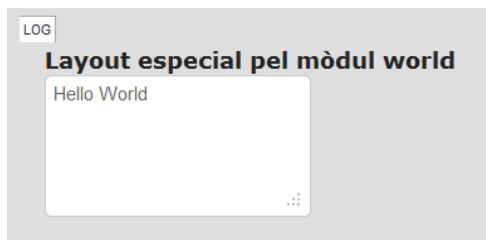
Actualitzem la pantalla d'administració i veurem que ens apareix el plugin que acabem de crear (si no apareix, esborrar cache abans), però aquest plugin no està activat. Tenim un botó per activar-lo amb el nom "Habilitar".



Fem clic i habilem el nou plugin que, de moment, no fa ni té res.



Ara farem que el plugin serveixi d'alguna cosa, per exemple, creant una vista pròpia del component input/longtext (tots els textareas es convertiran a botó). Recordem que aquest component l'hem utilitzat al layout *test* quan hem visualitzat el template *index* del mòdul *world*.



```
class WorldActions extends Actions
{
    public function executeIndex($request)
    {
        $this->content=Viewer::_echo('default:module').' World';
        $this->setLayout('test');
    }
}
```

Crearem llavors un fitxer a `/plugins/testPlugin/views/default/input/longtext.php` (cal crear abans les carpetes views, views/default i views/default/input) amb el següent codi:

```
<?php $js=isset($vars['js'])?$vars['js']:''; ?>
<button name="<?php echo $vars['internalname']; ?>" <?php echo
$js;?><?php echo $vars['value']; ?></button>
```

Com recordareu tenim la variable `$vars` que contindrà totes les variables que passem al component. Les variables que ja coneixem són *js* i *value*. Hi ha una nova variable que és comú a tots els components i es *internalname*. Aquesta variable conté el valor del nom que es donarà al component. Si no s'especifica s'assignarà un d'aleatori.

Incloem el nou plugin al mòdul o a l'aplicació, esborrem cache (per si de cas) i anem a la url <http://projectes/hello.php/world>. Que ha passat? Veurem que no ha passat res.

```
plugins:
- cubePlugin
- testPlugin
```

Prioritat del Plugin

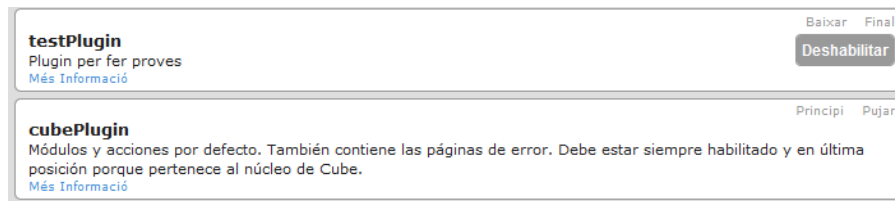
En l'apartat anterior ens em quedat amb les ganes de veure tots el textareas transformats a botó, però ara ho solucionarem.

Els plugins instal·lats a cube porten una prioritat. Aquesta prioritat s'assigna automàticament segons la data de inclusió al framework. Mitjançant l'administrador de plugins podem canviar aquesta prioritat.

La prioritat és una dada important ja que les vistes, i18n i mòduls depenen d'ella. En l'exemple anterior tractem amb la part de les vistes però el funcionament és equivalent per a la internacionalització i les configuracions dels mòduls.

Que ha passat abans? Que cubePlugin és més prioritari que testPlugin.

Encara que *testPlugin* estigui actiu, i inclòs a la nostra aplicació, el component *input/longtext* també existeix a *cubePlugin*, que és més prioritari que *testPlugin* per trobar-se més adalt. Haurem d'anar a l'administrador de plugins i modificar la posició d'un (només hi ha dos).



Anem a l'acció index del mòdul world i veurem com el component ha canviat:



Configuracions

Ja hem vist varies configuracions a l'exemple: les configuracions i18n (lang.xx_xx.yml) i les pròpies del projecte, aplicació i mòdul (config.yml). Així com per recollir els valors de i18n ho fem amb `Viewer::_echo`, per extreure un valor de les configuracions tipus config.yml ho farem amb el mètode **get** de la classe **Config**. Aquestes configuracions, per contra que les i18n, es poden modificar en temps d'execució amb el mètode **set** de la classe **Config**.

Cube, llegeix la configuració segons la ruta executada i deixa un array de valors a disposició de l'usuari per accedir a ells des de qualsevol punt de l'execució. Cal dir que aquestes configuracions van canviant a mesura que anem executant els filtres interns. Per exemple, primer es carrega la configuració del site, després la de l'aplicació, la del mòdul i finalment, la dels plugins actius per aquella ruta especificada. Anem a fer una prova:

Afegim una nova variable que defineix un color a l'aplicació (per exemple després de definir els plugins actius), i escriurem al template el codi següent, "Hello World" apareixerà en blau:

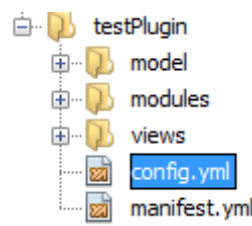
```
<div style="color:<?php echo Config::get('test_color');?>">
<?php echo $content; ?>
</div>
```

```
plugins:
- cubePlugin
- testPlugin
test_color: "#00f"
```

Fixeu-vos que el valor està entre cometes, això és perquè si no les posem, el parser de yml detecta el caràcter # i per tant comenta la línia. Canviem la configuració al mòdul i canviem test_color a #f00.

```
plugins:
test_color: "#f00"
```

Ara el text surt en vermell. Això és perquè la configuració del mòdul a sobreescrit la de l'aplicació. Provem de fer el mateix a testPlugin, afegim test_color: "#f0f" al config principal del plugin (recordeu que les configuracions dels plugins són les últimes en carregar-se):



Esperàvem que el text canviés a rosa però no ha sigut així. Perquè? Les configuracions pròpies del plugin porten intrínsecament el nom del plugin com a prefix, és a dir, si afegim *test_color* com a variable de configuració del plugin accedirem a ella mitjançant `Config::get('testPlugin:test_color');`

D'aquest fet es deriva que podem afegir variables a les configuracions dels plugins des de l'aplicació o mòdul si afegim el prefix del plugin. Hi ha plugins que necessitaran de variables que entre aplicacions canviaran. Imagineu un plugin que visualitza menus verticals. A cada aplicació podem definir el menú que volem que serà visualitzat pel plugin.

Si volem canviar el valor de `test_color` a l'acció ho farem amb `Config::set` :

```
public function executeIndex($request)
{
    Config::set('test_color', "#080");
    $this->content=Viewer::_echo('default:module').' World';
    $this->setLayout('test');
}
```

Ara si que visualitzem el text en verd. ☺



`Config::get` té un tercer paràmetre que correspon al valor per defecte. Igual que fèiem amb `$request->getParameter('clau', 'valorperDefecte')` també podem fer una cosa semblant amb `Config`, `Config::get('clau','all','valorperDefecte')`.

És important destacar que els mètodes `set` i `get` són estàtics, o sigui, que poden ser cridats des de la classe.

El segon paràmetre correspon al nivell on s'ha definit el paràmetre (site – principal, app – aplicació, module – mòdul, all - global). El valor 'all' permet recuperar el valor al primer nivell on s'ha definit, ordenat inversament, és a dir, mòdul – aplicació – principal. Si no es troba definit a cap dels 3 nivells retorna 'valorperDefecte'.

El següent codi mostra l'exemple de com accedir a la variable `test_color` del plugin:

```

public function executeIndex($request)
{
    Config::set('test_color', "#080");
    $this->content=Viewer::_echo('default:module').<span style="color:'.
        Config::get('testPlugin:test_color', 'all', '#ff0')."> World</span>;
    $this->setLayout('test');
}

```

En aquest punt la variable del plugin encara no ha sigut creada. El text 'World' es mostra en groc, que és el valor per defecte que hem posat:



Afegim ara la variable `test_color` al `config.yml` del plugin. El text queda:

```

all:
  security:
    all:
      is_secure: false
      test_color: "#f0f"

```

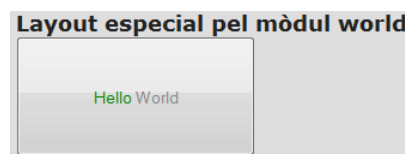


Fem la prova de crear `testPlugin:test_color` al mòdul. És tan fàcil com crear l'etiqueta `testPlugin` amb la subetiqueta `test_color`:

```

plugins:
  test_color: "#f00"
  testPlugin:
    test_color: "#888"

```



Fixeu-vos que la variable `test_color` que hem definit al `config.yml` de `testPlugin`, és la mateixa que `testPlugin:test_color` definida al `config.yml` del mòdul

Singletons

El patró de disseny Singleton és molt útil per no haver d'instanciar objectes d'una mateixa classe per fer algunes operacions. A més a més la gestió de memòria és més eficient ja que només es crea una instància de la classe a cada execució de l'acció.

El patró Singleton s'utilitza a Config, Controller, Request, Session, Site, User i View.

Hem vist que el paràmetre `$request` de l'acció correspon a un objecte de la classe Request. Com està implementat amb el patró Singleton, aquest objecte es pot accedir amb `Request::getInstance()` tenint accés a la mateixa informació des de diferents parts del codi. La variable `$request` es passa per a que sigui més ràpid accedir a aquest objecte i també per compatibilitat amb les definicions d'accions a symfony 1.X, per tant podem arribar a escriure el codi `Request::getInstance()->getParameter('clau', 'valorperDefecte')` a qualsevol part del codi (acció, vista – component o template, altres classes definides,...).

Més endavant veurem que fan les classes Session, Site i User.

Filtres

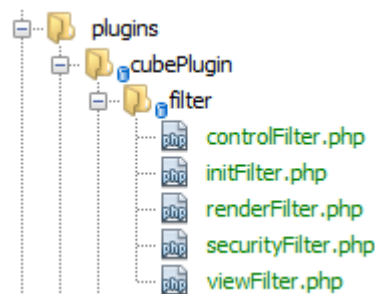
Els filtres d'execució representen els estats d'una execució d'una acció. Cube executa una sèrie de passos, separats pel tipus de servei que donen, de forma semblant a Symfony 1.X. Tenim definits cinc filtres per defecte: init, security, control, view i render. A la configuració principal (/engine/config.yml) estan definits així:

```
filters:
  init:
    class: initFilter
    package: plugins.cubePlugin.filter.initFilter
    enabled: on
  security:
    class: securityFilter
    package: plugins.cubePlugin.filter.securityFilter
    enabled: on
  control:
    class: controlFilter
    package: plugins.cubePlugin.filter.controlFilter
    enabled: on
  view:
    class: viewFilter
    package: plugins.cubePlugin.filter.viewFilter
    enabled: on
  render:
    class: renderFilter
    package: plugins.cubePlugin.filter.renderFilter
    enabled: on
```

Veiem que cada filtre porta una classe associada que està en un paquet concret, i que pot estar activat o desactivat. Podem definir els filtres d'aquesta manera o de manera més ràpida en forma d'array a l'aplicació i al mòdul ja que la informació de la classe i el paquet ja està definida a la configuració principal:

filters: [init, control, render] Això voldria dir que només estan actius els filtres init, control i render.

Com podeu comprovar aquests filtres estan definits a *cubePlugin*. Si aneu al sistema de fitxers veureu que el package coincideix amb l'arbre de fitxers:



Els filtres es defineixen com una classe amb el sufix Filter. Per exemple el filtre init és una classe *initFilter* que extén de *Filter* i té només un mètode *execute* que correspon a codi que s'executarà quan s'activi el filtre init (el primer de tots):

```

class initFilter extends Filter{

    public function execute($filterChain)
    {
        $i18n=Config::get("settings:i18n","site");
        Site::setLocaleConfig($i18n);
        $filterChain->execute();
    }
}

```

Aquest mètode té un paràmetre d'entrada *\$filterChain* que correspon a la cadena d'execucions de l'acció.

El filtre **init** només s'encarrega de preparar i configurar el sistema de internacionalització segons els paràmetres de la configuració principal:

- Agafem la configuració mitjançant **Config::get** i seleccionem els valors de *settings:i18n* del *config.yml* principal (engine/config.yml). Els valors que canviem a l'aplicació o al mòdul no tindran efecte, perquè aquesta configuració s'executa quan encara no sabem l'acció que anem a fer.
- Configurem el lloc declarant aquesta configuració amb el mètode **setLocaleConfig** de la classe Site, que és la que inicialitza tot el lloc. Si fem memòria al controlador de desenvolupament de l'aplicació **hello**, per exemple, teniem:

```

$site=Site::getInstance(array('app'=>'hello','debug'=>true,'cache'=>false));
$conf=$site->readConfiguration();
Controller::createInstance($conf)->init();

```

- Executem el següent filtre de la cadena d'execució.

Si volem canviar el filtre *initFilter* per un altre només cal modificar en les configuracions el package o la classe. L'etiqueta que porten els filtres a les configuracions no cal que coincideixi amb el nom del filtre, només és per anomenar-los dins de les configuracions.

La cadena de filtres segueix amb:

- **Security:** Carrega la configuració de seguretat de l'aplicació, del mòdul i dels plugins i com ja té la informació dels plugins actius, prepara també la configuració dels mateixos.
- **Control:** En aquest punt ja podem assegurar que l'acció es pot executar. L'executem i preparem les variables per a l'execució de la vista (sistema de templates)
- **View:** Si l'acció s'ha de decorar, es procedeix a la seva decoració
- **Render:** Acaba l'execució actualitzant variables temporals i de debug. Si estem en mode desenvolupament fa visible la consola web de debug.

Seguretat

El segon filtre que s'executa és el de seguretat. Aquest servei proporciona mecanismes de seguretat a les planes de l'aplicació. Si la seguretat està activada no podrem veure la plana, o aquesta serà redirigida a una altra. Aquesta acció la podem activar per tothom o només per usuaris que tinguin unes certes credencials.

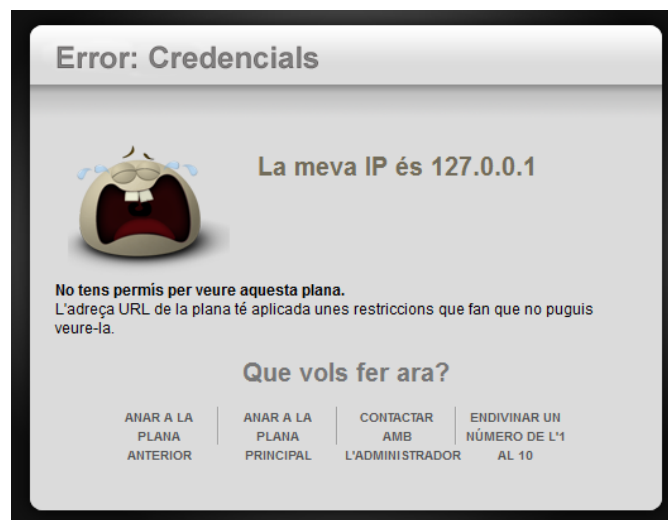
Als diferents fitxers de configuració hem vist una etiqueta **security** que tenia definits uns paràmetres:

```
security:
  all:
    is_secure: false
```

All denota totes les accions de l'aplicació (o el mòdul) i *is_secure* identifica aquella acció com a segura o no segura. Per defecte totes les accions són no segures, es a dir, tothom pot accedir a tot. Provarem de canviar *is_secure* a **true**. Que passa?

Cal tenir en compte en tot moment que la configuració del mòdul reescriu la de l'aplicació, per tant is_secure: true (també is_secure: on) l'hauré de posar al mòdul perquè l'acció index que estem executant durant totes les proves està al mòdul world.

No podem veure la plana perquè totes les accions del mòdul són segures.



Per defecte les accions segures es redirigeixen a una plana d'error però podem canviar aquest comportament afegint al codi una redirecció de la següent manera:

```
security:
  all:
    is_secure: on
    forward: default/error404Debug
```

El paràmetre *forward* que hem afegit redirigeix la plana a l'acció **error404Debug** del mòdul *default*. Recordem que *cubePlugin* té un mòdul *default*, que tenim actiu per defecte a totes les aplicacions. Ara la redirecció (que es fa internament, és a dir, no canvia la url) fa visualitzar:



Ara provarem de fer una altra acció que sigui segura i deixarem com a no segura l'acció *index*. Crearem l'acció *seguretat* al mòdul **world**. Les accions del mòdul ara són:

```
class WorldActions extends Actions
{
    public function executeIndex($request)
    {
        Config::set('test_color', "#080");
        $this->content=Viewer::_echo('default:module'). '<span style="color:'.
            Config::get('testPlugin:test_color', 'all', '#ff0'). '> World</span>';
        $this->setLayout('test');
    }

    public function executeSeguretat($request)
    {
        $this->content='Benvingut a la plana de seguretat';
        $this->setTemplate('index');
    }
}
```

Executarem la url http://projectes/hello_dev.php/world/seguretat. Com no hem canviat la configuració de seguretat visualitzarem la mateixa plana que l'anterior. Haurem de posar *is_secure* a **off**:



Però clar, aquesta acció es no segura. Com podem fer que l'acció sigui segura? Afegint-la com a nova etiqueta sota *security*:

```
security:
  all:
    is_secure: off
  seguretat:
    is_secure: on
```

Com podem llavors veure la plana anterior si la seguretat sempre està activada? Mitjançant credencials.

Sessió

Credencials

Les credencials són variables booleanes que identifiquen un atribut actiu o inactiu. Aquestes credencials es carregen a sessió, per tant, encara que les sessions de php no es carreguin automàticament es crea (o es reutilitza) una sessió per execució d'acció.

Com afegir i treure credencials? A qualsevol part del codi, tenint en compte que el filtre de seguretat es carrega després del filtre init. Per tant, a partir del filtre de control podem accedir i modificar les credencials.

Les credencials es troben a la sessió i per tant la gestió la farem des de la classe **Session**. Tenim cinc mètodes per treballar amb credencials:

- **getCredentials()**: recupera totes les credencials en forma d'array.
- **hasCredential('credencial')**: recupera el valor (true o false) d'una credencial específica
- **addCredential('credencial')**: afegeix la credencial 'credencial' a sessió. *setCredential* és equivalent (utilitzem addCredential per compatibilitat amb symfony 1.X).
- **removeCredential('credencial')**: esborra la credencial 'credencial' a sessió
- **removeAllCredentials()**: esborra totes les credencials de la sessió

Desactivem la seguretat de l'acció **seguretat** (*is_secure: false*) i canviem el codi de l'acció pel següent:

```
public function executeSeguretat($request)
{
    $this->content='<h1>Benvingut a la plana de seguretat</h1>';
    $this->content.='<br/>Credencials'._r(Session::getCredentials());
    $this->setTemplate('index');
}
```

Veiem que cridem al mètode estàtic *Session::getCredentials()*. Aquest ens ha de retornar un array de valors. Per visualitzar l'array utilitzem la funció *_r* (disponible en cube i no en php i utilitzada molt al codi, sobretot per depuració). Té un segon paràmetre (true), que visualitza el mateix array com un *var_dump*. Per últim veiem que el decorador de l'acció es canvia al template **index**. La pantalla presenta el següent format:

Benvingut a la plana de seguretat

Credencials

Constatem que no tenim definida cap credencial. Afegim la credencial **hello_credential**:

```
public function executeSeguretat($request)
{
    Session::addCredential('hello_credential');
    $this->content='<h1>Benvingut a la plana de seguretat</h1>';
    $this->content.='<br/>Credencials'._r(Session::getCredentials());
    $this->setTemplate('index');
}
```

Ara, la plana mostra:

Benvingut a la plana de seguretat

```
Credencials
Array
(
    [hello_credential] => 1
)
```

Veiem que el mecanisme funciona. Farem una última prova que demostrarà que el sistema de seguretat funciona: Afegirem la credencial a l'acció **index** i la treurem a l'acció **seguretat**. Això farà que només executant l'acció **index** abans que **seguretat** puguem visualitzar la pantalla anterior, i ademés només ho farem un cop. Caldrà fer l'acció **seguretat** com a segura canviant la configuració:

```
security:
  all:
    is_secure: off
  seguretat:
    is_secure: on
    credentials: [hello_credential]
```

El codi de les accions quedarà:

```
class WorldActions extends Actions
{
    public function executeIndex($request)
    {
        Config::set('test_color', "#080");
        $this->content=Viewer::_echo('default:module').'<span style="color:'.
            Config::get('testPlugin:test_color', 'all', '#ff0').'"> World</span>';

        $this->setLayout('test');
        Session::addCredential('hello_credential');
    }

    public function executeSeguretat($request)
    {
        $this->content='<h1>Benvingut a la plana de seguretat</h1>';
        $this->content.='<br/>Credencials'._r(Session::getCredentials());
        $this->setTemplate('index');
        Session::removeCredential('hello_credential');
    }
}
```

Fixeu-vos que encara que a la plana surti que tenim la credencial *hello_credential*, l'acció l'elimina després. Per tant la informació que tindrem és de l'estat de les credencials a l'iniciar l'acció. Fem la prova:

- executem la url http://projectes/hello_dev.php/world/index: això activarà la credencial.
- executem la url http://projectes/hello_dev.php/world/seguretat: el primer cop podrem entrar perquè la credencial existeix. Després serà esborrada.
- Tornem a executar la url http://projectes/hello_dev.php/world/seguretat: ara la credencial no existeix. Hi haurà una redirecció a una plana d'error.

El codi de la configuració del mòdul ha de quedar de la següent manera:

```
security:
  all:
    is_secure: off
  seguretat:
    is_secure: on
  credentials: [hello_credential]
```

L'acció *seguretat* és segura (*is_secure: on*), però els usuaris que tinguin les credencials *hello_credential* tenen permís per accedir, o dit d'una altra manera **només els usuaris amb la credencial *hello_credential* poden executar l'acció seguretat.**

La convenció que es fa servir per crear credencials complexes és molt semblant a la feta servir a symfony 1.X i és basa en el canvi de claudàtors []. Per cada nivell afegit hi ha un canvi d'AND a OR. Exemples:

[A,B]: A i B	[[A,B]]: A o B
[A,[B,C]]: A i (B o C)	[A,[B,[C,D]]: A i (B o (C i D))
[-A,B]: no A i B	-[A,B]: error, només negació de credencial

Variables de Sessió

Per treballar amb variables de sessió ho farem amb la classe Session i no amb \$_SESSION de PHP. De fet, estem treballant amb \$_SESSION però encapsulant-la amb una classe que conté més mètodes que ens són útils per treballar amb diferents variables com les abans esmentades com les credencials.

Els mètodes per gestionar les variables de sessió són:

- **Get**('variable','valorperDefecte'): agafa el valor d'una variable de sessió.
- **Set**('variable','valor'): defineix la variable de sessió amb un valor concret.
- **Un_set**('variable'): destrueix la variable de la sessió activa.
- **Is_set**('variable'): avalua si la variable de sessió existeix.

Variables Flash

Un altre tipus de variables de sessió són les variables flash. Aquestes variables tenen una peculiaritat i es que només estan disponibles durant una redirecció, es a dir, es destrueixen un cop la plana es carregada de nou. Això és molt útil a l'hora de visualitzar missatges d'error, que venen redirigits d'una plana de validació i només tenen sentit la primera vegada que la plana es recarrega. Així com les variables de sessió "pures" estan disponibles durant tota la vida de la sessió, les variables flash tenen un temps de vida d'una càrrega de contingut.

Anàlogament a les variables de sessió, els mètodes per treballar amb les variables flash són:

- **GetFlash**('variable','valorperDefecte'): agafa el valor d'una variable flash.
- **SetFlash**('variable','valor'): defineix la variable flash amb un valor concret.
- **Un_setFlash**('variable'): destrueix la variable flash de la sessió activa.
- **Is_setFlash**('variable'): avalua si la variable flash existeix.

Veiem un exemple de totes aquestes noves variables. Farem un comptador de visites a la nostra plana de seguretat, que només és visible si tenim la credencial *hello_credential*. Quan aquest comptador arribi a 5, destruïrem la credencial i ja no podrem accedir a la plana. El codi de les accions del mòdul quedarà de la següent manera:

[codi accions mòdul world]

Bàsicament, a l'acció **seguretat** comprovem si la credencial existeix, llavors incrementem el comptador de la variable de sessió. Si aquest és igual a 5, esborrem la credencial. L'acció **index** crearà la credencial i posarà la variable de sessió a 0. La primera vegada que executem **index**, creem la variable flash *hello_info* que ens dirà que comencem a contar i redirigirà a l'acció **seguretat**. Veuem que aquesta informació només la fa un cop ja que la variable flash es destrueix quan tornem a executar des de web l'acció **seguretat**.

Usuari

Per fer les coses més senzilles i tal i com ho fa symfony 1.X, hi ha definida una classe User que controla la validació i desconnexió d'un usuari a la sessió activa. De fet nosaltres utilitzem una classe MyUser que exten d'aquesta i implementa alguns dels seus mètodes abstractes.

MyUser treballa amb la sessió i fa servir tant les variables de sessió com les credencials per manipular el mecanisme de seguretat.

Hi ha definida la classe MyUser al model del plugin principal (cubePlugin) i està configurat per a treballar amb una validació virtual, es a dir, sense tenir en compte cap base de dades o directori actiu. És la definició més simple d'un usuari. Aquest model pot ser modificat directament al codi del cubePlugin (no recomanat) o mitjançant una definició del mateix model dins d'un plugin o d'una aplicació (recordeu que podem definir models a les aplicacions que només seran visibles a les mateixes).

El mètodes que té implementats, entre d'altres, són:

- **authenticateUser**: És el mètode encarregat de autenticar l'usuari al sistema (per base de dades, per ldap, ...). Retorna 0 per denotar que no hi ha hagut cap error. De fet hem simulat que ens hem autenticat correctament.
- **removeAllCredentials**: esborra totes les credencials de l'usuari
- **isLogged**: ens diu si l'usuari està validat al site.
- **executeLogout**: desconnecta l'usuari de la sessió, esborrant totes les credencials, destruint totes les variables de sessió i inicialitzant el flag de primera entrada al site.
- **executeLogin**: executa la validació de l'usuari. El sistema entén que l'usuari està validat al sistema i activa la credencial corresponent.

Com es pot veure, els mètodes de la classe MyUser utilitzen mètodes de la classe Session. Serà el mateix mirar MyUser::isLogged, que Session::hasCredential ('is_logged'). En el segon cas haurem d'enrecordar-nos com es deia la credencial que definia la validació al sistema.

Ara estem a punt de fer una aplicació amb contingut públic i privat. Tothom podrà executar l'acció **index**, però haurà d'estar validat per veure l'acció **llocPrivat**. Per això farem un miniformulari html que preguntí una clau d'accés mentre aquesta sigui incorrecta. Un cop

validats, el formulari reenviarà a la plana de seguretat amb la informació inicial “Usuari validat” i seguirà amb la lògica dels 5 intents per veure la plana. Al 5é intent desconnectarem l’usuari de la sessió.

Model

Bla bla bla

Formularis

Bla bla bla

Generadors

Bla bla bla

Cube 2

Bla bla bla