



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1



Harbour for Web 



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Preámbulo	3
Mercury	4
Configuración de nuestro sistema	5
App(). Empezando.....	6
TRoute	7
TController	9
TRequest	12
TValidator	13
TResponse.....	14
TMiddleware.....	14
View (Vistas)	15
A.- Macrosustitución.....	16
B.- PRG	18
B.1.- Recodida por número de parámetro	20
B.2.- Recogida de variables via un Setter/Getter	21



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Preámbulo

Me gustaría comentar un par de cosas con todos vosotros antes de empezar. Somos un grupo de jóvenes que llevamos una temporada trabajando en ese mundillo de locos. Muchos venimos de sistemas antiguos, de Clipper, luego Harbour.

Por motivos profesionales uno tiene de saltar a otros sistemas dada la evolución tecnológica que tenemos el privilegio de vivir y a mi me tocó hacer también muchos cambios.

Harbour para mi es sin duda el lenguaje mágico de mi vida, es especial. Llevaba tiempo sin programarlo pero siempre lo he tenido allí, a mi lado. Después de varios encuentros y reuniones con muchos de vosotros una de las grandes cuestiones que se abordaron era que Harbour se quedó atrás, obsoleto, triste. Nos faltaba el gran paso, dar el salto a la Web, como todos los lenguajes populares que existen hoy en día.

Mr. Antonio Linares se lo sacó una vez mas de la chistera y lo consiguió. La primera semilla del mod_harbour ya estaba puesta. ¿ Funcionará ?

Muchos lenguajes han pasado, pero Harbour sigue. Somos los últimos en subir al tren de la web. Muchos van delante, todos. Pero nosotros somos rascadores de code con “decenas” de años de experiencia y ahora es el momento de sacarnos todo nuestro conocimiento y aportarlo a la comunidad. Hemos de dar el salto y mirar de tu a tu a los otros. Acabamos de empezar pero ya hablamos con la bestia.

Ahora solo falta poner estos primeros cimientos y base a nuestro querido Harbour. Que la gente se anime y vea que poco a poco se va haciendo realidad. Creo que estamos viviendo un nuevo punto de inflexión y va a ser apasionante.

He intentado construir Mercury viendo como los demás lo hacen, los grandes, los que tiene éxito, intentando imitar la manera en que hoy están trabajando millones de personas en todo el mundo. Es una base, está claro y no llega ni al 1% de la potencia de los grandes frameworks, pero quiero ver si nuestro Harbour lo aguanta y podemos empezar a correr.

No sé si se usará, quizás a muchos ni les funcione, pero intento contribuir de alguna manera en nuestra comunidad aportando mi experiencia e intentar animar a otros muchos a hacerlo más grande.

Chicos... (jóvenes), Harbour es magico ☺. Os animo a probar, participar y disfrutar de nuestro nuevo Harbour.

Charly Aubia - 2019





Mercuy

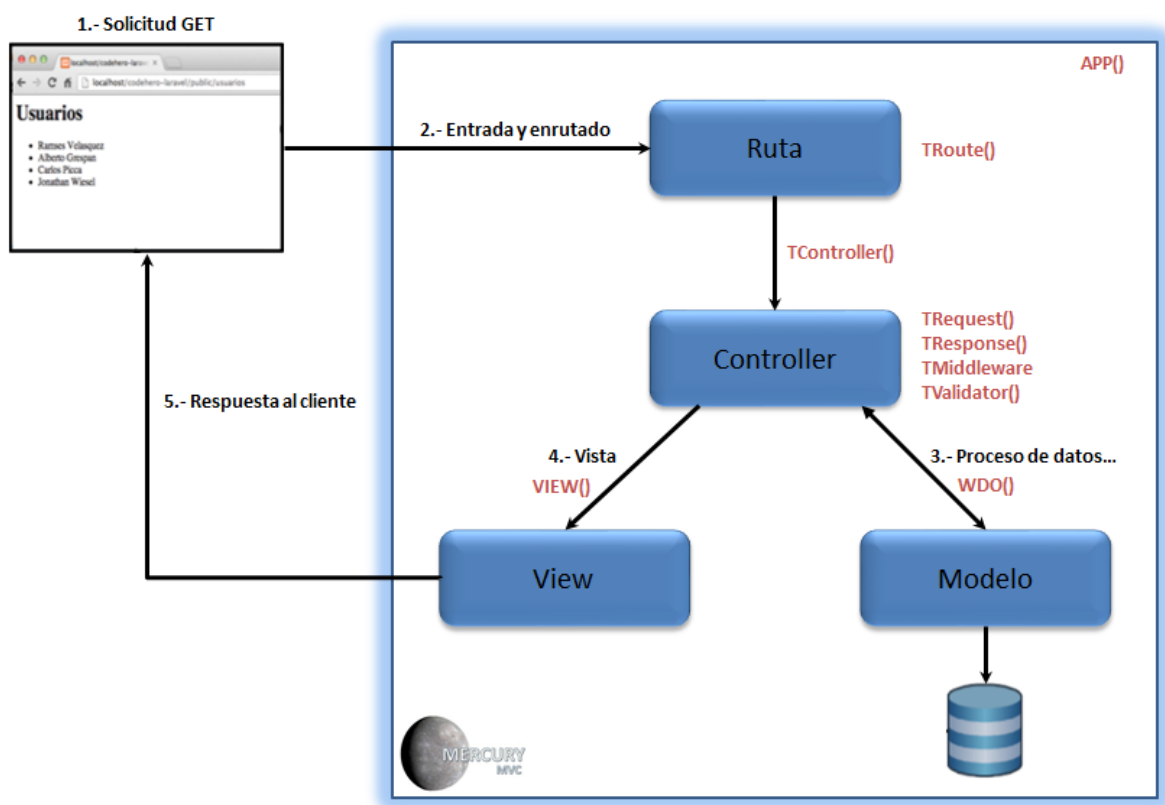
Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Mercury

Mercury es un motor de aplicaciones MVC que forma parte del mod_harbour y nos permitirá estructurar de una manera eficiente el diseño de una aplicación web hecha con Harbour.

Siguiendo los estándares de los grandes frameworks, tomaremos sus conceptos de diseño para aplicarlo a nuestro código fuente y para ellos usaremos las clases que nos provee Mercury.



Para poder usar Mercury, al inicio del programa habremos de especificar la carga del módulo de la siguiente manera:

```
//      {% LoadHRB( '/lib/core_lib.hrb' ) %}      //      Funciones core
//      {% LoadHRB( '/lib/tmvc_lib.hrb' ) %}      //      Sistema MVC Mercury
```

Esto es todo !

Asumimos que las librerías las pondremos en el directorio /lib



Configuración de nuestro sistema

El sistema MVC siempre tiene un punto de entrada por las URL. Eso significa que si nuestra app esta situada en localhost/hweb/apps/miapp cada vez que hagamos localhost/hweb/apps/miapp/miprog.prg, localhost/hweb/apps/miapp/folder/test2.prg, localhost/hweb/apps/miapp/folder1/foldere2/test3.prg, ... nuestro sistema hará un acceso por index.prg que será definido en nuestro sistema con el fichero .htaccess

```
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)$ index.prg/$1 [L]
</IfModule>
```

También es importante definir las siguientes variables de entorno que nos servirán de momento para la gestión de paths en la aplicación. Partiendo de la base de que tenemos nuestra aplicación en la carpeta /hweb/apps/minimvc, quedaría así:

```
SetEnv PATH_URL          "/hweb/apps/minimvc"
SetEnv PATH_APP          "/hweb/apps/minimvc"
SetEnv PATH_DATA         "/hweb/apps/minimvc/data/"
```

Estos nos permitirá fácilmente cambiar los paths del sistema.

Podemos definir los demás parámetros según nuestra conveniencia, pero un fichero base de .htaccess para nuestro propósito inicial podría ser el siguiente:

```
# -----
# CONFIGURACION RUTAS PROGRAMA (Relative to DOCUMENT_ROOT)
# -----
SetEnv PATH_URL          "/hweb/apps/mvc_mercury_hrb"
SetEnv PATH_APP          "/hweb/apps/mvc_mercury_hrb"
SetEnv PATH_DATA         "/hweb/apps/mvc_mercury_hrb/data/"

# -----
# Impedir que lean los ficheros del directorio
# -----
Options All -Indexes

# -----
# Pagina por defecto
# -----
DirectoryIndex index.prg main.prg

<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)$ index.prg/$1 [L]
</IfModule>
```



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

App(). Empezando...

Empezaremos el diseño de la aplicación con la declaración de la función App() que nos devolverá un objeto de la clase TApp(). Esta clase se encarga de iniciar el sistema, paths, funciones de seguimiento, logs, errores... . Creará internamente diferentes clases que posteriormente podremos usar en el diseño, como un router, request, response,... los iremos viendo a medida que los necesitemos.

Finalmente y antes de finalizar nuestro programa, ejecutaremos el metodo :Init() que iniciará todo nuestro sistema.

Así pues, tenemos que el programa empieza y acaba con la inicialización de TApp(). Podria quedar asi:

```
// -----  
// {% LoadHRB( '/lib/core_lib.hrb' ) %}           // Funciones core  
// {% LoadHRB( '/lib/tmvc_lib.hrb' ) %}           // Sistema MVC Mercury  
// -----  
  
FUNCTION Main()  
  
    LOCAL oApp    := App()  
  
    ...  
    ...  
    ...  
  
    //    Iniciamos el sistema  
  
        oApp:Init()  
  
RETU NIL
```

Ya tenemos armado la base del fichero index.prg, pero aún no hace nada.



TRoute

Como todos sabeis la base del MVC para por escuchar las peticiones que recibe la web y procesar esta petición enrutando (dirigiendo) la petición a un lugar u otro. Es aquí donde vamos a crear nuestras rutas de nuestro sistema y será en el fichero index.prg.

Para crear las rutas usaremos el objeto oRoute que se encuentra en oApp. El objeto oRoute tiene los siguientes métodos

Map(<método>, <id>, <mapa>, <controller>)

Mapear una ruta

<metodo>

GET/POST/PUT/DELETE

(De momento sólo soporta GET/POST)

<id>

Identificador del mapeo. Para usar desde cualquier punto de nuestra aplicación y buscar una ruta lo haremos por el id

<mapa>

Máscara de nuestra url. Estará formado por una parte fija y otra variable que podrán ser parámetros por defecto. Si nuestra url base es localhost/hweb/apps/minimvc , podremos añadir a continuación las diferentes opciones que permitimos para acceder a nuestra aplicación. P.e. imaginemos que queremos crear una entrada para consultar una lista y queremos que el parámetro se llame list. El mapa sera "list" . La url quedaria: localhost/hweb/apps/minimvc/list y esto provocaria una reacción en nuestro sistema ejecutando el controlador que especifiquemos. Si queremos por ejemplo consultar un usuario indicando ademas su número de id, este id seria un parámetro variable. Si queremos llamar a esta accion p.e. userinfo y al identificador id el mapa seria: "user/(id)" . El sistema sabe que podremos hacer una entrada p.e. de este tipo localhost/hweb/apps/minimvc/user/123.

Todos los parámetros variables irán entre paréntesis.

Para indicar una acción a nuestra url base el mapa será "/"

<controller>

Nombre del controlador que se ejecutará cuando se enrute la petición. Nuestro sistema cargará el programa que definamos. Un controller será un clase con varios métodos. Imaginemos que tenemos un controlador que se encarga de procesar 2 acciones: listar (list), consultar id (info) y este controlador lo llamamos Vecinos.

La estructura base de este controller seria p.e.

```
CLASS Vecinos
```

```
    METHOD New() CONSTRUCTOR
```

```
    METHOD list()
```

```
    METHOD info() // Menu de la aplicacion
```

```
ENDCLASS
```

```
METHOD New( o ) CLASS Vecinos
```



Mercuy Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

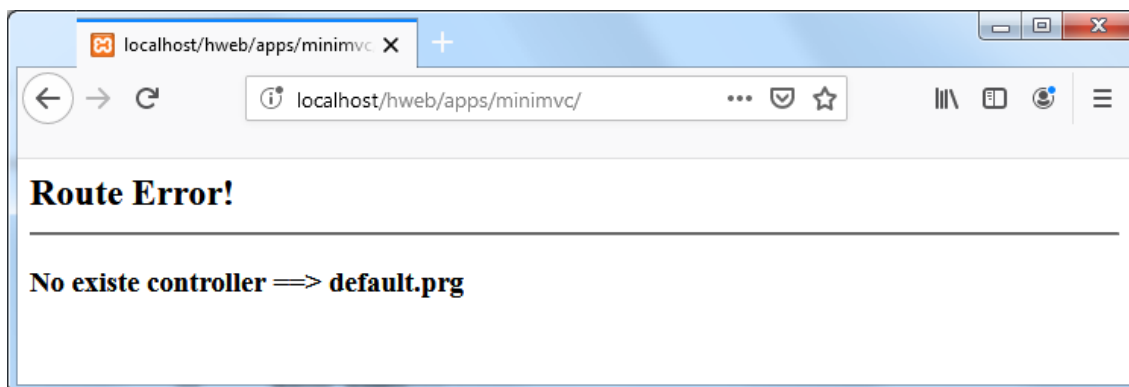
```
RETU SELF  
  
METHOD list( o ) CLASS Vecinos  
  
RETU NIL  
  
METHOD info( o ) CLASS Vecinos  
  
RETU NIL
```

Quando definamos en el Map el controller, este ira precedido de @ y el nombre de método que queremos que se ejecute. Imaginemos que creamos una ruta que si especificamos vecinos/list queremos ejecutar el metodo list de nuestro controlador vecinos. Habriamos de especificar "[list@vecinos.prg](#)"

Creando nuestro primer ejemplo en el sistema, crearemos una ruta que en el caso de que no introduzcan nada en nuestra url, ejecutaremos un controlador por defecto y ejecute una pantalla inicial. Nuestro controlador se llamará default y nuestro método welcome. La definicion seria asi:

```
oApp:oRoute:Map( 'GET' , 'default' , '/' , 'welcome@default.prg' )
```

Si ejecutamos nuestra aplicación el sistema nos mostrará el siguiente error, siempre que oApp:IShowError este a .T. (valor por defecto)



Nuestro sistema no encuentra el controlador default.prg. Todos los controladores se encontrarán por defecto en la carpeta /src/controller



TController

Crearemos nuestro controlador default.prg con el siguiente code

```
CLASS Default

    METHOD New() CONSTRUCTOR

    METHOD Welcome()

ENDCLASS

METHOD New( o ) CLASS Default

    RETU SELF

METHOD welcome( o ) CLASS Default

    ? '<h1>Welcome to my page !</h1>'

    RETU NIL
```

Y volvemos a refrescar nuestro navegador



Y ya tenemos nuestra página y hemos avanzado en el concepto

En este ejemplo desde el propio método del controlador imprimimos un mensaje con ? que procesa una salida de un string por la pantalla, pero, por norma, las salidas se generarán con el objeto oResponse o le diremos al objeto oController:View(<cFile>) que redireccione a una vista y ya se encargará de generar la salida. Pero para efectos prácticos y comprobación rápida, podemos usar el ?.

Ya veremos más adelante el tema de las View, pero en el caso de que no deseemos generar una salida tipo *vista*, con oResponse dispondremos de métodos que nos facilitaran la salida con otros formatos.



Mercuy Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Los métodos de un controlador siempre recibirán como parámetro un objeto oTController, con las siguientes datas:

oController	Descripción
oRequest	Objeto TRequest. Nos servirá para examinar todas las variables referentes a la entrada. Ver TRequest
oResponse	Objeto Tresponse. Nos servirá para generar salidas. Ver Tresponse
oMiddleware	Objeto TMiddleware. Nos servirá para crear nuestra seguridad de acceso al controlador. Ver TMiddleware
cAction	Acción que envía el enrutador
hParam	Hash de parámetros que envia el enrutador
:ListController()	Información del Controlador. A efectos de <i>debugueo</i> , ver lo que recibimos.
:ListRoute()	Lista de Rutas definidas en index.prg. A efectos de <i>debugueo</i> .
:View(<cFile>, ...)	Ejecutar la vista <cView> pasandole en el caso de haberlos, los parametros para usarlos desde la vista

Podríamos crear 2 métodos dentro de default prg para poder consultar la información que le llega y las rutas con los métodos ListRoute() y ListController().

```
METHOD Info( oTController ) CLASS Default

    oTController:ListController()

RETU NIL

METHOD Routes( oTController ) CLASS Default

    oTController:ListRoute()

RETU NIL
```

Y crear las rutas en el index.prg

```
//      Basic pages...
oApp:oRoute:Map( 'GET', 'default'      , '/'      , 'welcome@default.prg' )
oApp:oRoute:Map( 'GET', 'routes'      , '?'      , 'routes@default.prg' )
oApp:oRoute:Map( 'GET', 'info'        , 'info', 'info@default.prg' )
```

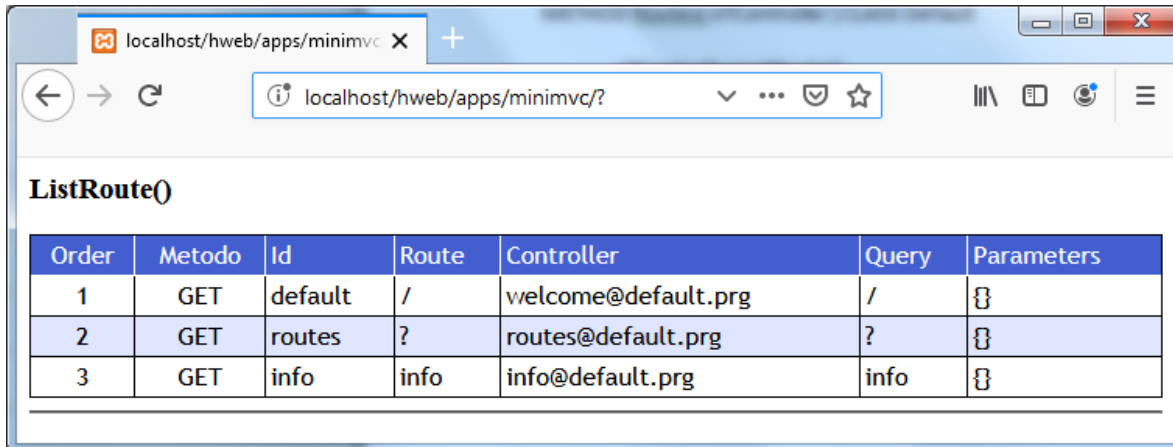


Mercuy

Modelo/Vista/Controlador

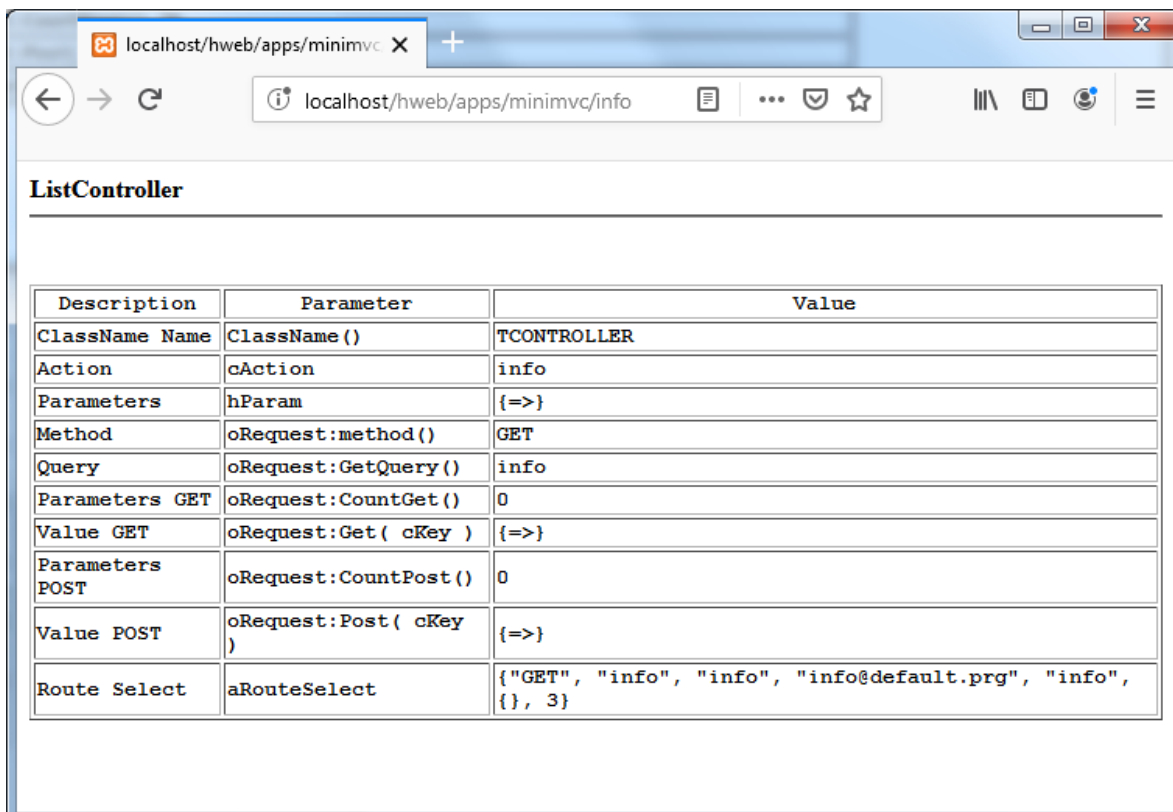
Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Ejecutamos <http://localhost/hweb/apps/minimvc/>?



Order	Metodo	Id	Route	Controller	Query	Parameters
1	GET	default	/	welcome@default.prg	/	{}
2	GET	routes	?	routes@default.prg	?	{}
3	GET	info	info	info@default.prg	info	{}

Ejecutamos <http://localhost/hweb/apps/minimvc/info>



Description	Parameter	Value
ClassName Name	ClassName()	TCONTROLLER
Action	cAction	info
Parameters	hParam	{=>}
Method	oRequest:method()	GET
Query	oRequest:GetQuery()	info
Parameters GET	oRequest:CountGet()	0
Value GET	oRequest:Get(cKey)	{=>}
Parameters POST	oRequest:CountPost()	0
Value POST	oRequest:Post(cKey)	{=>}
Route Select	aRouteSelect	{"GET", "info", "info", "info@default.prg", "info", {}, 3}



Mercury Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

La parte de controller es una de las más delicadas. Es la parte del procesamiento de la petición y a grandes rasgos recibimos el input de datos que lo gestionaremos con el objeto oRequest, procesaremos datos usando en muchas ocasiones los modelos de datos y pasaremos a las Vistas (View) los datos para que “pinten” la pantalla

Cuando se ejecute el método del controller, hemos visto que recibimos el oController. Lo primero que hemos de hacer es mirar los parámetros que recibimos en la petición. Por ejemplo si me piden los datos de un cliente, es necesario recibir al menos un parámetro que identifique a este cliente, imaginemos que es id.

Como hemos diseñado nuestro sistema, sabremos que método se usará: get, post o ambos,

El objeto oController ofrece 3 métodos directos para acceder a estos datos en función del metodo:

```
oController:GetValue( 'id', 0, 'N' )  
oController:PostValue( 'id', 0, 'N' )  
oController:RequestValue( 'id', 0, 'N' )
```

```
oController:xxxxValue( <cKey>,[<cValueDefault>], [<cFormat>] )
```

<cKey> Nombre del parámetro
<cDefault> Valor por defecto. Por defecto es un cadena vacia
<cFormat> Si queremos ya formatear el datos. Pueden ser los típicos tipos de variable de harbour: 'C', 'N', 'D'

TRequest

Con estas funciones ya podremos recuperar perfectamente los datos, pero si queremos más detalle podemos acceder a ellos con los métodos directos del objeto oController:oRequest

oRequest - Métodos	Descripción
Method()	Metodo por la que se ha hecho la petición
Get(cKey, uDefault, cType)	Recuperar valor via GET
GetAll()	
CountGet()	
Post(cKey, uDefault, cType)	Recuperar valor via POST
PostAll()	
CountPost()	
Request(cKey, uDefault, cType)	Recuperar valor via REQUEST
RequestAll()	
GetQuery()	Recuperar de la Query de la Url
GetUrlFriendly()	
GetCookie(cKey)	Recuperar Cookie



TValidator

Una vez recuperado el parámetro lo correcto seria validar la entrada de dato: si tiene el formato esperado, longitud, valores permitidos,... Para este propósito usaremos la clase Tvalidator(). La clase validator nos validará los parámetros que hemos recuperado en función de unas reglas. Por ejemplo, imaginemos que en el caso del id queremos que sea un parámetro obligatorio y que sólo entren numeros. Las reglas irán en un hash y podremos especificar tantas como queramos y en este caso seria algo asi:

```
LOCAL hRoles      := { 'id' => 'required|numeric' }
```

Crearemos un objeto Tvalidator

```
LOCAL oValidator   := Tvalidator():New()
```

Y el concepto es: Voy a validar una/s variable con una/s reglas. Si todo OK continuamos dentro de nuestro Controller, sinó daremos una respuesta al cliente. Esta respuesta podra ser una vista u otro tipo de respuesta (que ya veremos mas adelante) como un json,xml,... Si se produce un error, la clase Tvalidator nos devolverá por el método :ErrorMessage() un array con todos los errores que se han encontrado. La técnica habitual es p.e. si no se pasa la validación se envia a una vista con los errores y la vista, si lo deseamos, ya mostrará estos errores.

Asi pues, nos quedaría el sistema de validación así:

```
//      Validacion de datos

IF ! oValidator:Run( hRoles )
    oRequest:View( 'default.view' ,{ 'success' => .F, 'error' => oValidator:ErrorMessage() } )
    RETU NIL
...

```

Aquí podremos engancharnos al tema de respuestas via oResponse que comentabamos anteriormente. Imaginemos que estamos creando una api y la respuesta en lugar de una página html, queremos devolverla en json. En este caso usariamos el metodo oResponse:SendJson() para generar esta salida

```
//      Validacion de datos

IF ! oValidator:Run( hRoles )
    oRequest:oResponse:SendJson( { 'success' => .F, 'error' => oValidator:ErrorMessage() } )
    RETU NIL
...

```



TResponse

oResponse - Métodos	Descripción
SetHeader(cHeader, uValue)	Crear una cabecera de salida
SendJson(uResult, nCode)	Crear cabeceras JSON y salida de datos. nCode default = 200
SendXml(uResult, nCode)	Crear cabeceras JSON y salida de datos. nCode default = 200
SendHtml(uResult, nCode)	Crear cabeceras JSON y salida de datos. nCode default = 200
Redirect(cUrl)	Redireccionar a Url
SetCookie(cName, cValue, nSecs)	Generar una cookie

TMiddleware

Los Middleware serán los guardas y protectores de nuestros controllers. Los podremos usar según convenga y el objetivo es verificar la autorización de la petición a ejecutar un método

Básicamente pondremos el middleware en el método NEW() del controlador. Allí podremos chequear el Middleware() de tipo JWT para saber si un usuario está autenticado. De momento es el que tenemos activado, ya que aún no tenemos sesiones en marcha.

El concepto es: Si ejecuto el middleware() en el controller y no lo pasa lo podemos redirigir a una vista, p.e. la pantalla default, de login,... Si el middleware lo autoriza el controlador ejecutará el método de la petición.

La construcción sería algo así

```
METHOD New( o ) CLASS MyApp

    //    Control de acceso al controlador. El middleware se aplicará a todos los métodos
    //    excepto a los que indiquemos aquí

    IF o:cAction $ 'default'           //    Módulo que NO se validan: públicos, defaults,...

        // 'No middleware...'

    ELSE

        o:Middleware( 'jwt', 'boot/default.view' )           //    View

    ENDIF

RETU SELF
```

El middleware lo podremos recuperar del objeto oController:oMiddleware, p.e

```
LOCAL oMiddleware    := oController:oMiddleware
```



Mercuy Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

El middleware de tipo 'jwt' es el que usaremos cuando un usuario se autentique. Tendremos un controlador para validar el acceso. En el caso de que se autentique correctamente ejecutaremos el método:

```
oMiddleware:SetAutenticationJWT( hTokenData, nTime )
```

Donde hTokenData son datos que llevara el token de validación y siempre podremos recuperar en cada petición y nTime el tiempo de validación de este token entre cada petición. Cada vez que se realiza una petición el token se regenera de nuevo con el tiempo definido.

Si un usuario se sale del sistema y genera un logout ejecutaremos el método:

```
oMiddleware:CloseJWT()
```

View (Vistas)

La parte de las vistas es el último proceso en el que finalmente hemos de coger los resultados y "pintarlos" con html. Ya hemos comentado antes que es posible que desde la parte controller hagamos una contestacion al cliente con oResponse en un formato de datos como p.e. Json y esto no pinta datos, sino que devuelve una respuesta y listos, pero este punto de las views va dirigido a como montamos la salida en html

El objeto oRequest tiene un metodo :View(<cFileView>, ...), que será el encargado de cargar nuestro fichero con html, procesarlo y devolverlo al navegador cliente. Los ficheros <cFileView> van por defecto ubicados en la carpeta /src/view por lo que los ubicaremos allá con una extension.view, por la que llamada desde el controller podria ser asi oController:View('home.view', cName).

El fichero contendra html puro y duro y no hay mas secreto que el uso standard de html,js,css...

```
<html>

    <h2>Hello Vista !</h2>

</html>
```

La particularidad de estos *.view es que podemos mezclar nuestro código para optimizar su uso y crear nuestro proceso de creación y diseño de una página con el uso de 2 técnicas de programación:



A.- Macrosustitucion HFW.

En el que indicaremos una función encerrada entre {{ ... }} → {{ time() }} en la que el sistema procesará todas las que tenga el fichero antes de devolverlo. Un ejemplo sencillo seria:

```
<html>

    <h2>Hello Vista !</h2>

    Ahora son las {{ time() }} del dia  {{ date() }}

</html>
```

Podremos usar todas las funciones Harbour a nuestra disposición para poder componer nuestra web !!!

Tenemos tambien el uso de una funcion especial que es View(<file.view>). Esto aun nos da mas flexibilidad a nuestras paginas. Imaginemos que tenemos una cabecera de html que en todas las paginas hace lo mismo: declarar pagina html, cargar css, ficheros js,... p.e.

```
<!DOCTYPE html>
<html>
<head>
    <title>Titulo de mi App</title>

    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <link rel="shortcut icon" type="image/png" href="favicon.ico' } }"/>

    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">

    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js"></script>
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js"></script>

</head>
```

Imaginemos que tenemos 3 paginas normales: page1.view, page2.view, page3.view . Estas 3 paginas han de tener tambien definida su cabecera.El código anterior lo pongo en un fichero head.view. y ahora podríamos contruir las 3 páginas así:



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Test1.view

```
{{ View( 'head.view' ) }}  
  
    <h2>Hello Vista 1 !</h2>  
  
    Ahora son las {{ time() }} del dia  {{ date() }}  
  
</html>
```

Y esto en los 3 ficheros. Listos !!!

Ahora imaginemos que tenemos la típica barra de bootstrap, un nav en las 3 paginas con este code

```
<nav class="navbar navbar-dark bg-dark">  
    <a href="https://google.com"></a>  
    <a class="navbar-brand" href="hweb/apps/minimvc/index.prg">Home</a>  
  
    ...  
  
</nav>
```

Podríamos meter este mini code (template) en un fichero → nav_default.view

Ahora, en las 3 páginas que hemos creado añadirlo de la siguiente manera:

Test1.view

```
{{ View( 'head.view' ) }}  
{{ View( 'nav_default.view' ) }}  
  
    <h2>Hello Vista 1 !</h2>  
  
    Ahora son las {{ time() }} del dia  {{ date() }}  
  
</html>
```

Y listos: 3 paginas con este tipo de code. Cuando modifiquemos head.view o nav_default.view automáticamente afectará a las 3 paginas. Este es el concepto de los template que bien combinado nos dará más productividad y eficiencia en nuestros códigos.



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Sólo faltaría complementar esta parte con uso correcto de los path para cargar nuestros recursos. Si tenemos este code:

```
<!DOCTYPE html>
<html>
<head>
  <title>Titulo de mi App</title>

  ...

  <link rel="shortcut icon" type="image/png" href="../images/favicon.ico' } }"/>

  ...

  <script src="../js/public.js' } }"></script>

  ...
</head>
```

¿ Recordáis nuestra variable de inicio oApp := App() ? Podemos hacer referencia en cualquier punto de nuestro programa con App() y con ella acceder a muchos datos de nuestra aplicación. Podríamos aprovecharla para definir nuestro code de esta manera:

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ App():cTitle }}</title>

  ...

  <link rel="shortcut icon" type="image/png" href="{{ App():Url() +
'/images/favicon.ico' } }"/>

  ...

  <script src="{{ App():Url() + 'js/public.js' } }"></script>

  ...
</head>
```

App():Url() siempre nos dara el path base de nuestra aplicación para poder usarlo de referecnia en nuestras definiciones.

B.- PRG

Esta es la otra opcion de trabajar en las views. Se trata de insertar código Harbour entre todo el html separado por los tags <?prg y ?>.





Mercuy Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Podemos crearnos nuestras rutinas de proceso de datos pero al final hemos de devolver un código de html que se insertará y sustituirá al código preprocesado, p.e..

```
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h2>Test View<hr></h2>

    <?prg

        LOCAL cHtml := ""
        LOCAL nI

        cHtml += "<ul>"

        FOR nI := 1 TO 4

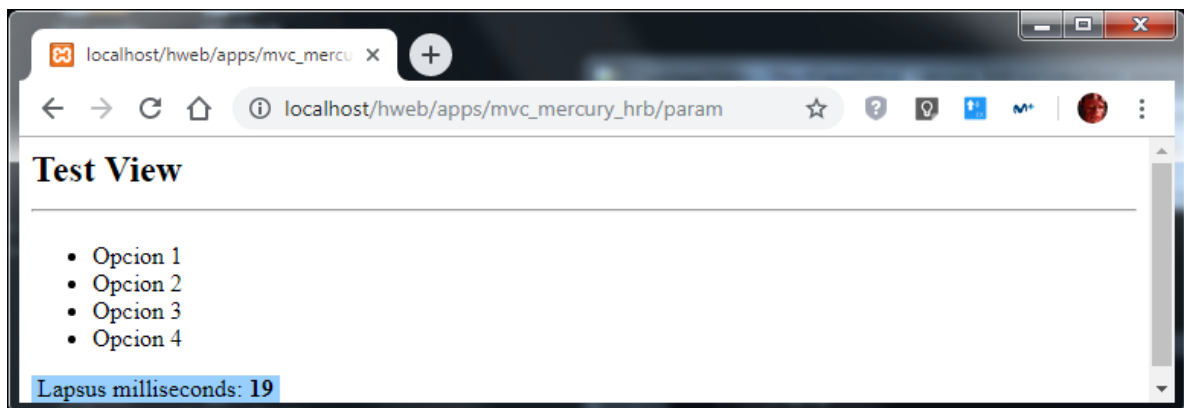
            cHtml += "<li>Opcion " + ltrim(str(nI))

        NEXT

        cHtml += "</ul>"

        RETU cHtml

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



Recogida de parámetros desde el Controlador

Como hemos explicado, el controlador se encarga de procesar datos y luego los pasa a la vista para que los use. Tenemos 2 maneras para hacerlo



Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

B.1.- Recogida por número de parámetro. Nosotros sabemos cuando diseñamos el controlador que cuando llamamos al metodo view le podemos pasar parámetros, pe.

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge

    //    Tratamiento de datos

        cName := 'Maria de la O'
        nAge  := 47

    //    Solicitud de Vista

        o:View( 'test_param2.view', cName, nAge )

    RETU NIL
```

cName es el parametro 1, nAge el 2, ...

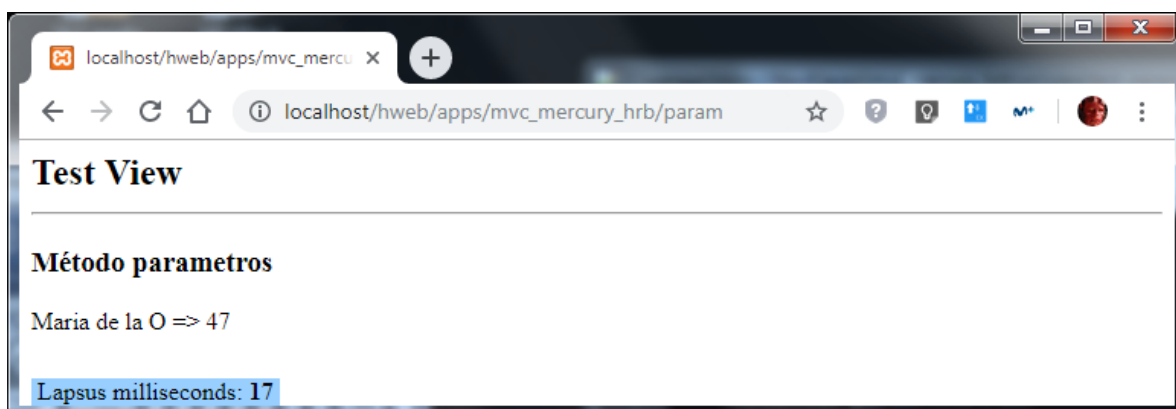
En la vista podemos hacer referencia a ellos de esta manera

```
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <h2>Test View<hr/></h2>

  <h3>Método parametros</h3>

  {{ PARAM 1 }} => {{ PARAM 2 }}

</body>
</html>
```





Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

B.2.- Recogida de variables via un Setter/Getter, pe. en el controlador especificar los datos que queremos usar en la vista con un setter → `_Set(<cVar>, <uValue>)`

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge, aData

    //    Tratamiento de datos

    cName := 'Maria de la O'
    nAge  := 47
    aData := { 'manzana', 'pera', 'cereza', 'platano' }

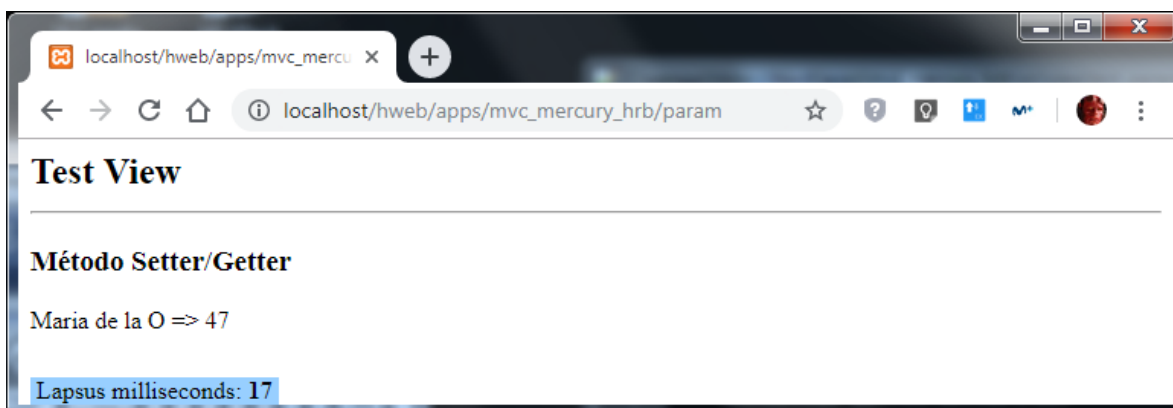
    //    Solicitud de Vista

    _Set( 'name' , cName )
    _Set( 'age'  , nAge  )
    _Set( 'fruit', aData )

    o:View( 'test_param2.view' )

RETU NIL
```

Y en la vista usar la funcion `_Get(<cVar>)` para recoger esta variable





Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

Vamos a usar todo junto para poder valorar diferencias

Código del controller

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge, aData

    //    Tratamiento de datos

    cName := 'Maria de la O'
    nAge  := 47
    aData := { 'manzana', 'pera', 'cereza', 'platano' }

    //    Solicitud de Vista

    _Set( 'name' , cName )
    _Set( 'age'  , nAge )
    _Set( 'fruit', aData )

    o:View( 'test_param2.view', cName, nAge )

RETU NIL
```

Código de la view

```
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
    <h2>Test Parameters: _Get()<hr></h2>

    <h3>Método parametros</h3>

    {{ PARAM 1 }} => {{ PARAM 2 }}

    <br><br>

    <h3>Método Setter/Getter</h3>

    {{ _Get( 'name' ) }} => {{ _Get( 'age' ) }}

    <?prg

        LOCAL aData    := _Get( 'fruit' )
        LOCAL cHtml    := '<hr><h3>Fruits..</h3>'
        LOCAL nI

        FOR nI := 1 TO len( aData )

            cHtml += '<li>' + aData[ nI ]

        NEXT

        RETU cHtml

    ?>

</body>
```

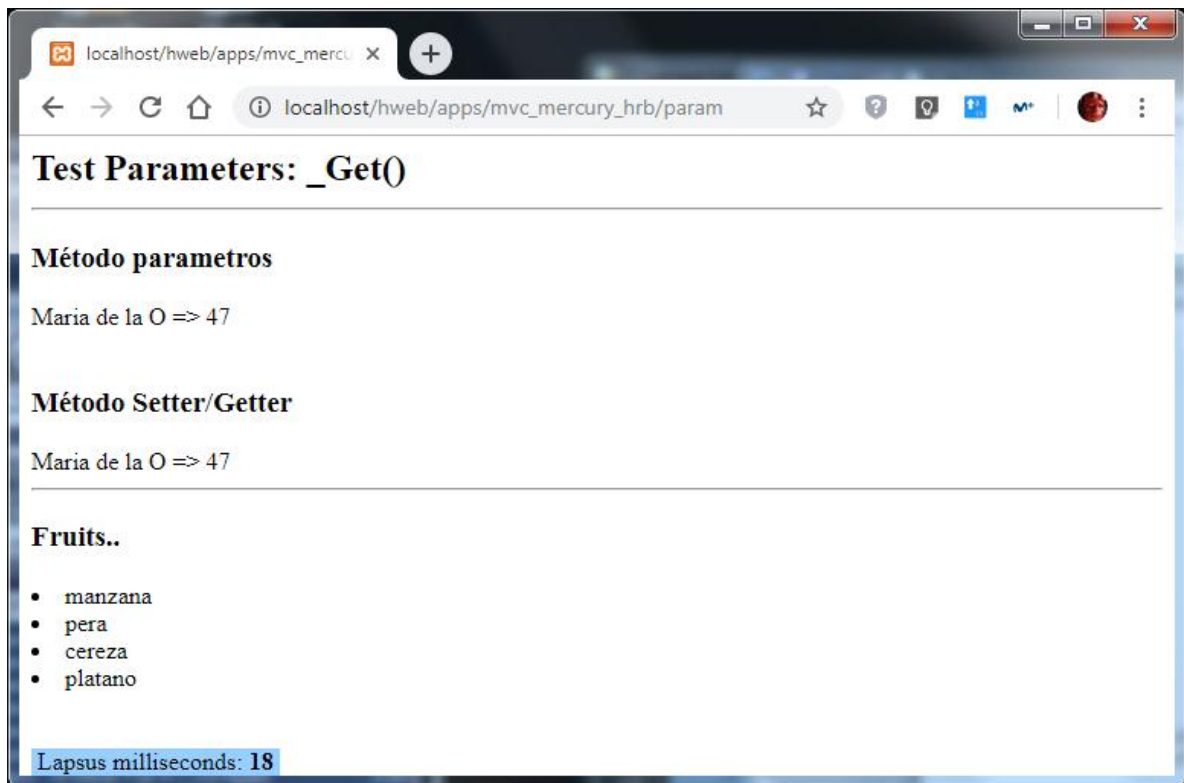


Mercuy

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 0.1

</html>



Y esto es todo !!! El uso correcto de diferentes templates junto al uso de la macrosustitucion HFW y código prg (embedido), nos dará una tremendo rendimiento, fiabilidad y facilidad de mantenimiento a nuestras app.

MODEL – WDO

Not Yet !!!