



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02



Harbour for Web
mod Harbour





Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Preámbulo	4
Mercury.....	5
Como usar este manual.....	5
Instalación.....	6
Configuración de nuestro sistema.....	7
1.- Activando el mod_rewrite de Apache.....	7
2.- Configurando nuestro fichero .htaccess	7
Creando nuestro primer PRG	9
Estructura de ficheros.....	9
Fichero principal del proyecto → index.prg.....	10
¿Porqué MVC?.....	11
1ª Parte - Creando nuestro primer proyecto.....	12
Router: Index.prg	12
View	14
Nuestra primera vista: hello.view	14
Inyectando código PRG.....	18
Inyectando código HTML dentro de PRG.....	19
Controller.....	22
a.- Ejemplo de un controlador que envia los datos a una view.....	23
b.- Ejemplo de un controlador que envío los datos al navegador	27
Creación de una view formulario para buscar el state.....	29
Recogida de parámetros desde el Controller.....	31
Objeto oRequest.....	31
Recoger parámetro y buscar en la tabla	33
Pasar el resultado a la view	34
Modelo	35
Creando el Modelo Customer.....	36
2ª Parte - Avanzando en el patrón.....	41



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

TValidator.....	41
Autenticación.....	45
Login.....	46
Controller de Validacion.....	47
Welcome	49
Middleware	50
Resumen final de nuestro sistema.....	52
3ª Parte – Técnicas avanzadas.....	53
View() - View de View.....	53
Css().....	57
Maquetado.....	60
WebServices.....	64
Funcionamiento básico del WS	65
Método GetByState() – Listado de clientes por estado	65
Método Auth() - Creación de sistema autenticación.....	68
Método New() - Validación del token.....	71
Método Data() – Ver datos incrustados en el token.....	73
Conclusiones.....	76



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

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 como xBase, 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 saco una vez más de la manga y lo consiguió. La primera semilla del mod_harbour ya estaba puesta. Funcionará ?

Muchos lenguajes han pasado, pero Harbour sigue. Somos los últimos de 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 emular la manera en que hoy están trabajando millones de personas en todo el mundo. Cuando lo empecé era otra semilla en este proyecto, pero después de un año ha madurado lo suficiente para darnos confianza en este sistema y un camino para empezar con nuestros proyectos de manera eficaz.

Durante este año también he podido constatar lo difícil que resulta a la mayoría entrar en este entorno de programación y sobre todo adaptarse a esta nueva arquitectura del software, pero porque venimos de otra cultura de programación y sabemos que nos hemos de reciclar. Se que los cambios siempre cuestan.

Este manual lo he reeditado para hacerlo mas fácil de seguir y todos los ejemplos están listos para probar. Empezamos de 0, paso a paso, quizás muy fácil para quien conozca el sistema, quizás muy difícil a medida que avancemos, pero acabamos con un sistema de autenticación, control de acceso a los módulos y toda la aplicación cerrada.

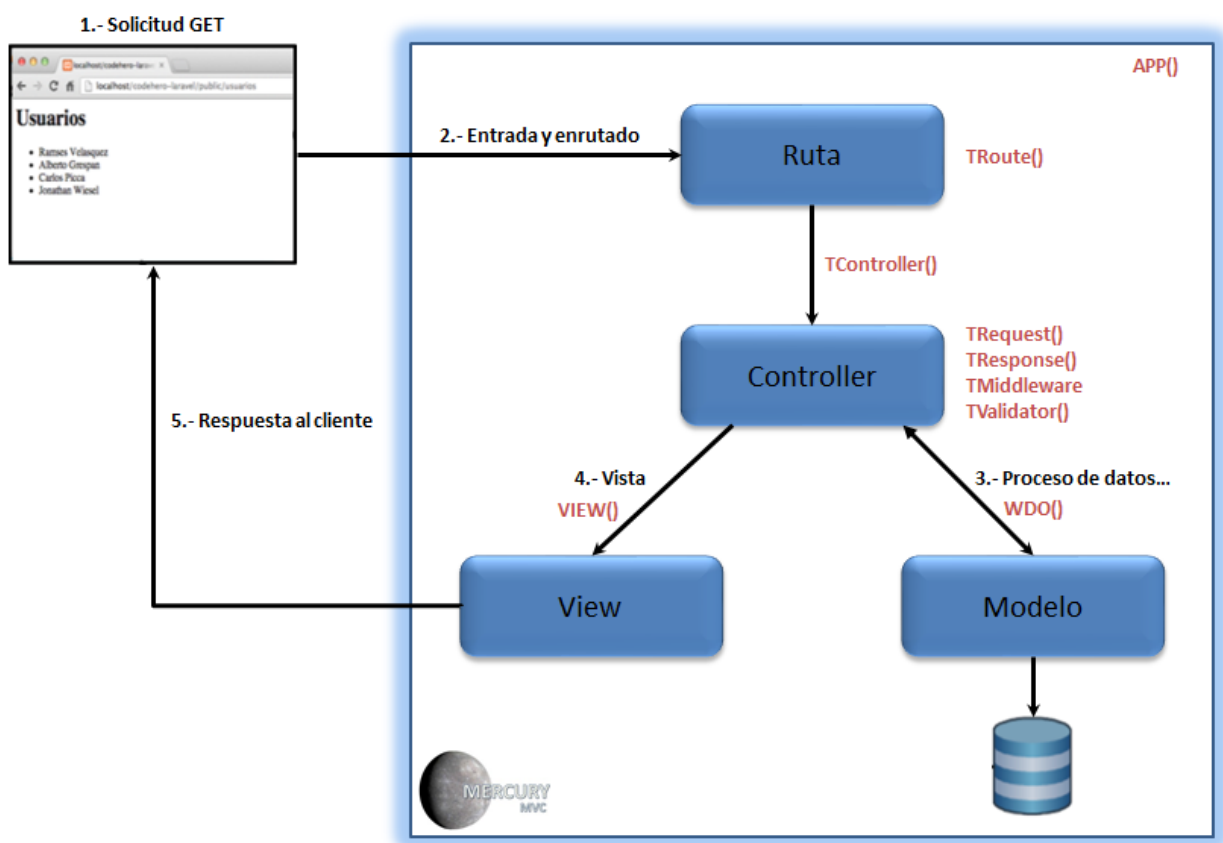
Colegas... Harbour es mágico ☺. Os animo a probar, participar y disfrutar de Mercury para crear vuestras aplicaciones web y dar el paso definitivo a la web.



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 se intenta seguir la misma línea y conceptos a la hora de diseñar el programa.



Iremos explicando paso a paso todos estos conceptos, el porqué, las ventajas, el encaje de una pieza con otra...

Como usar este manual.

La mejor manera de aprender es rascando y probando los ejemplos. Todos han sido probados y lo mejor es seguir paso a paso y hacer copy/paste e ir probando y entender el por qué de lo que estamos haciendo. Si lo realizamos hasta final del manual, os aseguro que entenderéis y podréis empezar a hacer vuestras aplicaciones.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Instalación

Partimos de la base, que ya tenemos instalado mod Harbour y funcionando correctamente, sino podéis consultar en esta página como hacerlo → <https://modharbour.app/compass/search/instalacion>

Lo primero que debemos hacer es bajar el framework desde <https://github.com/carles9000/mercury.git> haciendo un download en zip

The screenshot shows the GitHub repository page for 'carles9000 / mercury'. The repository has 63 commits, 1 branch, 0 packages, 0 releases, and 2 contributors. A 'Clone or download' button is visible, and a dropdown menu is open showing options to clone with HTTPS, SSH, or download a ZIP file. The repository files listed are css, data, doc, and images.

O quien use git hacer un clone → `git clone https://github.com/carles9000/mercury.git`



Configuración de nuestro sistema

1.- Activando el mod_rewrite de Apache

Hemos de parametrizar Apache para que podamos usar el módulo rewrite. Para ello iremos al fichero de configuración httpd.conf y ver que tengamos la línea del módulo activa (descomentada)

```
161 #LoadModule request_module modules/mod_request.so
162 #LoadModule reqtimeout_module modules/mod_reqtimeout.so
163 LoadModule rewrite_module modules/mod_rewrite.so
164 #LoadModule sed_module modules/mod_sed.so
```

Y añadir estas líneas

```
DocumentRoot "C:/xampp/htdocs"
<Directory "C:/xampp/htdocs">
    Options Indexes FollowSymLinks Includes ExecCGI
    AllowOverride All
    Require all granted
</Directory>
```

2.- Configurando nuestro fichero .htaccess

Recordad apagar y volver a iniciar el server para que estas líneas tengan efecto. ¿Que se consigue con estos cambios? Podemos redirigir las peticiones que se hagan a nuestro servidor a un punto de entrada. La arquitectura MVC siempre tiene el mismo punto de entrada. Eso significa que si nuestra app está situada en localhost/miapp cada vez que hagamos localhost/hweb/miapp/miprograma.prg, localhost/miapp/folder/test2.prg, localhost/miapp/folder1/folder2/test3.prg, ... nuestro sistema redirigirá el acceso a localhost/miapp/index.prg.

Para acabar de conseguir esto, en nuestro proyecto definiremos el fichero .htaccess el cual nos sirve para configurar el entorno de nuestra aplicación. En el añadiremos estas líneas

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



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

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 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/"
```

Esto nos permitirá fácilmente si algún día cambiamos de ubicación ajustar 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/minimvc"  
SetEnv PATH_APP "/hweb/apps/minimvc"  
SetEnv PATH_DATA "/hweb/apps/minimvc/data/"  
  
# -----  
# Impedir que lean los ficheros del directorio  
# -----  
Options All -Indexes  
  
# -----  
# Pagina por defectos  
# -----  
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>
```



Resumen: Una vez tengamos configurado nuestro apache con el punto 1, esta parte ya no lo tocaremos mas. Para cada nuevo Proyecto ajustaremos los paths de nuestro fichero .htaccess como se muestra en el punto 2.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Creando nuestro primer PRG

Estructura de ficheros

Partimos de la base que ya tenemos instalado modHarbour y esta corriendo perfectamente. El método que usaremos es válido tanto en entornos Windows como Linux, pero nos basaremos en el entorno Windows por ser más los usuarios que lo usáis. Tengáis solo Apache instalado o Xampp, crearemos nuestro directorio para el proyecto dentro del directorio htdocs, p.e. go → /htdocs/go y crearemos la siguiente estructura

/htdocs/go/ /lib/mercury/mercury.hrb /lib/mercury/mercury.ch /include /src/controller /src/model /src/view	Directorio App web Biblioteca Mercury Fichero de cabecera para usar mercury Directorio donde tendremos ficheros include de harbour Directorio donde pondremos los controladores Directorio donde pondremos los modelos Directorio donde pondremos los views
--	---

Y finalmente en la raíz crear nuestro fichero .htaccess → /htdocs/go/.htaccess que tendría esta configuración

```
# -----  
# CONFIGURACION RUTAS PROGRAMA (Relative to DOCUMENT_ROOT)  
# -----  
SetEnv PATH_URL          "/go"  
SetEnv PATH_APP          "/go"  
SetEnv PATH_DATA         "/go/data/"  
  
# -----  
# Impedir que lean los ficheros del directorio  
# -----  
Options All -Indexes  
  
# -----  
# Pagina por defectos  
# -----  
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>
```

Esta es nuestra estructura básica de nuestro programa. A medida que avanzaremos iremos añadiendo otros directorios si los necesitamos, como el data, include,...



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

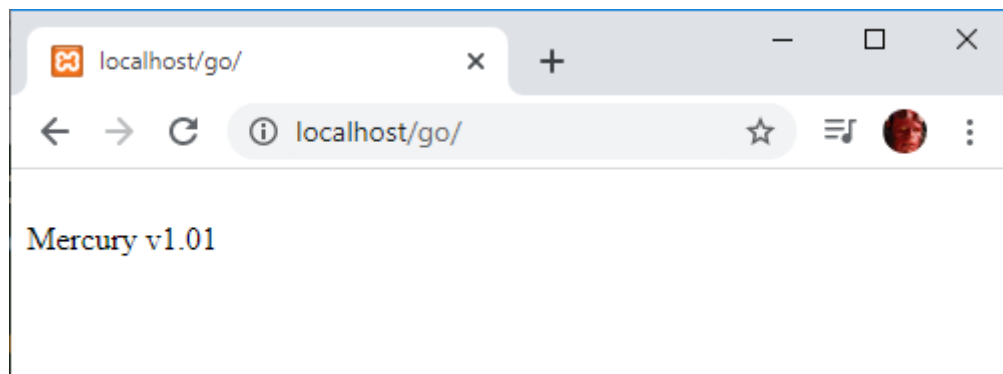
Ya solo nos queda probar si todo está en orden. La mejor manera para comprobarlos es crear un programa que nos cargue el *mercury* y nos escriba en pantalla su versión. Si llegamos a este punto significará que todo está en orden

Fichero principal del proyecto → [index.prg](#)

Nuestro fichero `index.prg` será el punto de entrada a nuestra aplicación. Lo primero que debemos hacer es comprobar que tenemos *mercury* bien instalado. Crearemos el `index.prg` y añadiremos las siguientes líneas:

```
//      {% LoadHRB( 'lib/mercury/mercury.hrb' ) %}      //      Loading system MVC Mercury  
  
function Main()  
  
    ? MercuryVersion()  
  
retu nil
```

Si vamos a nuestro navegador y escribimos `localhost/go` nos deberá aparecer una pantalla similar a esta



Es muy importante ver la primera línea que es donde cargamos nuestra librería *mercury*, y que la sintaxis es

```
//      {% LoadHRB( '/lib/mercury/mercury.hrb' ) %}
```

Es necesario poner las `//` de comentarios, porque apache ejecutará antes de iniciar nuestro programa todo lo que se encuentre entre `{% y %}`. Es una manera de realizar la carga de módulos `hrb` y que funciona perfectamente.

Esto es todo !



¿Porqué MVC?

Creo que la entrada en la Wikipedia está bastante bien explicado y definido:

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

La ventaja de usar este sistema es que una vez asimilado obtendremos grandes beneficios en nuestros programas y nuestros mantenimientos serán mejores. Sabremos donde tendremos que ir en cada momento y tendremos todo bien ordenado y estructurado.

Hay una curva de aprendizaje que nos puede costar asimilar bastante a los que venimos de otros entornos de programación, pero creo que es el mejor sistema para salir con éxito de un proyecto web. Al final podremos afirmar que este modelo sigue el "divide y vencerás".

Al inicio veremos que todo lo troceamos en muchos ficheros y no entendemos como pudiendo realizar en un único fichero todas nuestras necesidades, tenemos de crear quizás 3,4,5... Pero veremos que todo encaja cuando lleguemos al final del manual.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

1ª Parte - Creando nuestro primer proyecto

Como he explicado antes existen un montón de opciones a la hora de diseñar nuestra aplicación y lo más importante es entender como encajaran las distintas piezas. Como hemos comentado, cada petición que hacemos a nuestra app pasará por un `index.prg` que será la página de entrada a nuestra app.

En el `Índex` definiremos nuestra aplicación, nuestro mapa, nuestro Router, que será el encargado de despachar las distintas peticiones a nuestra aplicación. Todo lo que no esté definido en este mapa no se podrá ejecutar y en resumen todos los diferentes accesos los programaremos desde aquí.

Router: Index.prg

Empezaremos con la estructura del fichero `index.prg`, donde iniciaremos la carga de *mercury*, la definición de la aplicación y creación de nuestras rutas

```
// -----  
// Title.....: Hello !  
// Description: Example de web application with mercury...  
// Date.....: 22/05/2020  
// -----  
// {% LoadHRB( 'lib/mercury/mercury.hrb' ) %} // Loading system MVC Mercury  
// -----  
  
#include {% MercuryInclude( 'lib/mercury' ) %}  
  
function Main()  
  
    local oApp  
  
    // Define App  
    DEFINE APP oApp TITLE 'My web application...'  
  
    // Config Routes  
    DEFINE ROUTE 'root' URL '/' VIEW 'hello.view' METHOD 'GET' OF oApp  
  
    // System init...  
    INIT APP oApp  
  
return nil
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Esta es la base de nuestro fichero maestro index.prg. Vamos a describir las partes más importantes

- Carga de nuestra librería mercury.hrb

```
//      {% LoadHRB( 'lib/mercury/mercury.hrb' ) %}
```

- Carga del fichero de cabecera para poder usar comandos del Mercury

```
#include {% MercuryInclude( 'lib/mercury' ) %}
```

- Definición de nuestra Aplicación con el comando DEFINE APP

```
DEFINE APP oApp TITLE 'My web application...'
```

- Creación de Rutas. Es la parte donde definimos los distintos accesos a nuestros módulos desde nuestra url. El comando es el siguiente:

```
DEFINE ROUTE <cRoute> URL <cUrl> CONTROLLER <cController> METHOD <cMethod> OF <oApp>  
DEFINE ROUTE <cRoute> URL <cUrl> VIEW <cView> METHOD <cMethod> OF <oApp>
```

Parámetro	Descripción
cRoute	<p>Es un identificador que le damos a nuestra route. Este identificador lo podremos usar desde otros puntos de la aplicación para hacer referencia a su definición. Por ejemplo, si tenemos definido una ruta así:</p> <pre>DEFINE ROUTE 'pedido' URL 'order' CONTROLLER do@orders.prg METHOD 'GET' OF oApp</pre> <p>En cualquier punto del programa podremos ejecutar la función Route('pedido') y esta me devolverá la url definida, en este caso “order”</p>
cUrl	<p>Es la url definida para esta ruta. Siguiendo la definición anterior, si en la url de nuestro navegador ponemos “order” → localhost/go/order , nuestra aplicación sabrá que tenemos de ejecutar en este caso el controlador do@orders.prg , porque lo tenemos definido que se pueda ejecutar usando el meodo GET (ver cMethod)</p> <p>Si la url la definimos como '/' le está diciendo a nuestra aplicación que si no entramos nada en nuestra url del navegador, ejecute lo que tengamos definido → localhost/go</p> <pre>DEFINE ROUTE 'hello' URL '/' VIEW 'hello.view' METHOD 'GET' OF oApp</pre>
cController	<p>Será nuestro controlador, nuestro prg que se encargar de ejecutar el código pertinente para esta acción. Estos prg serán pequeñas clases, con sus métodos que invocaremos desde el</p>



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

	Router. En este caso le decimos a la aplicación que ejecute nuestro controlador orders.prg e invoke al método do()
cView	<p>Será nuestra vista asociada a esta ruta. A veces no necesitamos ningún controlador que procese datos, simplemente mostrar una view.</p> <p>DEFINE ROUTE 'hello' URL 'hola' VIEW "hello.view" METHOD 'GET' OF oApp</p> <p>Si ponemos en nuestra url 'hola' → localhost/go/hola , nuestro sistema sabrá que tendrá de ejecutar nuestra view 'hello.view'</p>
cMethod	<p>Es el método GET o POST que se usan en html para definir un tipo de acceso y que podremos usar. Hay mucha documentación sobre el tema https://es.stackoverflow.com/questions/34904/cuando-debo-usar-los-m%C3%A9todos-post-y-get</p> <p>Básicamente cuando queramos cargar paginas usaremos el método GET, mientras que el POST se usa mas para envío y actualización de datos.</p> <p>Atención si queremos ejecutar una Url, pues tendremos que definir en el route que es de tipo 'GET'. En el caso de que queramos definir una ruta que se ejecute en cualquiera de los casos podemos añadir ... METHOD "GET,POST"</p>
oApp	Objeto APP que hemos definido previamente

- Al final del index.prg iniciaremos la app con el comando INIT APP <oApp>



Resumen: Estamos creando una base donde definimos todos los diferentes accesos a nuestra aplicación, un mapa. Esto implica que a medida que vaya creciendo nuestra aplicación iremos sumando distintas rutas en nuestro index.prg

View

Nuestra primera vista: hello.view

La view será nuestra parte de código que defina la página que se va a mostrar en nuestro navegador, nuestro html. Nos guste o no, la web tiene sus lenguajes que deberemos usar de una manera u otra por lo que es necesario un conocimiento al menos básico para entender su funcionamiento. Básicamente usaremos html, css y javascript, por lo que es importante explorarlos 😊

Las view estarán en el directorio src/view y podremos añadir carpetas a ese directorio por si las necesitamos. Uno de los objetivos del MVC es seguir una jerarquía y ordenar todo, para que luego nuestros mantenimientos sean más fáciles.





Mercury

Modelo/Vista/Controlador

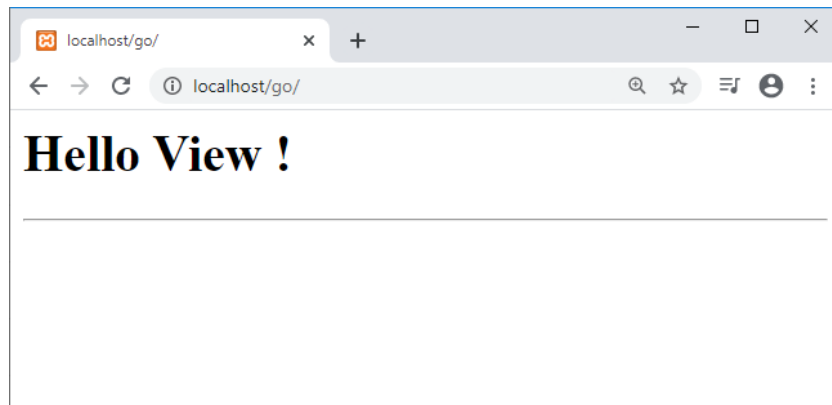
Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Siguiendo nuestro ejemplo, crearemos el fichero `src/view/hello.view` y pondremos el siguiente code

```
<h1>Hello View !</h1>
<hr>
```

Ya tenemos una vista creada. Html puro y duro. Serán páginas en las que la codificación será en html, no en prg, aunque mas adelante veremos como insertar código prg en nuestras views.

Pues ya tenemos todo necesario para ejecutar nuestra primera aplicación MVC en modHarbour. Si escribimos en la url de nuestro navegador → `localhost/go` nos debería aparecer



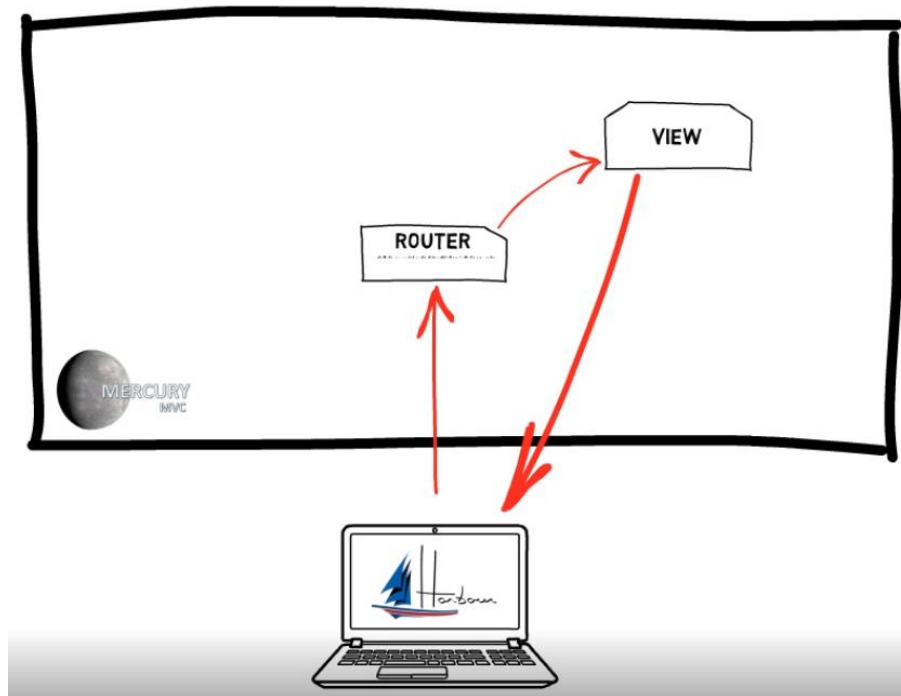
Ya tenemos nuestra base para ir escalando nuestra aplicación y mantener todo el control.



Resumen:

- Creación de una ruta que apunte a nuestra vista
 - o `DEFINE ROUTE 'root' URL '/' VIEW 'hello.view' METHOD 'GET' OF oApp`
- Definición de nuestra view → `hello.view`

Conceptualmente tendríamos este escenario → https://youtu.be/N6fN_4pIdj4



La primera pregunta que nos podemos hacer es porque tanta historia sin con este código también lo podemos hacer

```
function main()
  ? "<h1>Hello View !</h1>"
  ? "<hr>"
return nil
```

Debéis pensar que estamos diseñando los cimientos para nuestra aplicación, en la que el estricto orden y control será necesario y que, al ir escalando, nuestras funcionalidades no serán problema. MVC es una arquitectura del software que está muy probada y aceptada en la comunidad de desarrolladores de las principales plataformas existentes. Si entendemos este concepto podremos continuar con nuestro propósito.

Para acabar este primer ejemplo vamos a crear una simple página que nos muestre un link hacia otra página, para ver cómo manejar nuestro enrutador. La segunda página tendrá otro link que nos mandará de nuevo a la primera.

Una de las ventajas de programar html, css, js,... es que tenemos millones de páginas de ayuda, o sea q no estaremos solos en esta batalla. Por ejemplo, para buscar cómo crear un link, podemos encontrar un ejemplo de aquí → https://www.w3schools.com/tags/tag_a.asp



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Así pues crearemos 2 páginas que llamaremos view1.view y view2.view y definiremos 2 rutas en nuestro index.prg siguiendo el procedimiento descrito.

```
DEFINE ROUTE 'root'    URL '/'      VIEW 'hello.view' METHOD 'GET' OF oApp
DEFINE ROUTE 'view1'   URL 'view1'  VIEW 'view1.view' METHOD 'GET' OF oApp
DEFINE ROUTE 'view2'   URL 'view2'  VIEW 'view2.view' METHOD 'GET' OF oApp
```

Observad como le decimos a nuestra aplicación lo que podemos usar y lo que no, es decir, en estos momentos podemos poner en la url go/, go/view1, go/view2. Cualquier otra entrada será ingnorada

Ahora definiremos la view1 de la siguiente manera para poder ver como “*enrutaremos*” hacia la view2

```
<h1>Hello View 1</h1>
<hr>

<a href="{{ Route( 'view2' ) }}">Visit View 2 !</a>
```

Observamos el primer concepto de macrosustitución en nuestro código html usando las llaves {{ ... }}. El sistema sustituirá el código harbour que tengamos entre las llaves. En este caso tenemos nuestra función Route(<cRoute>) que es la nos devolverá la url asociada.

A función Route() le dice a nuestro sistema cual es la url que tenemos definida bajo la ruta “view2” y según tenemos definido en nuestro router esa es “view2.view”. El sistema hará lo que tenga que hacer pero al final devuelve go/view2 . Eso quiere decir que al final el código pre-compilado acabaría así:

```
<h1>Hello View 1</h1>
<hr>

<a href="go/view2">Visit View 2 !</a>
```

El sistema siempre se guiará por lo que definimos en nuestro Router. Nosotros no nos debemos preocupar donde está el código ni direcciones url, ni nada. Ya se encargara el Route de montarlo. A medida que vaya creciendo nuestra aplicación, si tenemos p.e. 5 views que hacen referencia a una url, haciéndolo de esta manera, si algún día cambiamos en el route la dirección, automáticamente cambiará en las 5 views, no nos deberemos preocupar de cambiar en cada vista manualmente esa nueva url, esta es una de las muchas virtudes de trabajar de esta manera.

Continuando con el ejemplo, crearemos la view2 de la siguiente manera.

```
<h1>I'm View 2</h1>
<hr>

<a href="{{ Route( 'view1' ) }}">Come back to View 1 !</a>
```



Es el mismo ejemplo, pero en esta página hacemos referencia a la view1



Y este es el concepto en este apartado del Router. Si lo probáis veréis que saltáis de un lado a otro y es nuestro router quien se encargará de controlar estas urls.

Este punto es importante porque debemos entender como desde los distintos puntos de nuestro programa, cuando hagamos una petición al server, la manera de encontrar estas urls ha de ser a partir de la definición de nuestro mapa de la aplicación que hemos creado en index.prg, nuestro "router".

Inyectando código PRG

Dentro de una view la cual estamos codificando en html, podemos inyectar código prg. La manera de hacerlo es abriendo la etiqueta `<?prg` y cerrando `?>` Dentro podremos poner tanto código harbour como queramos, la única condición es que devuelva un string que se insertará justo donde empieza la etiqueta.

```
<h1>Test PRG</h1>
<hr>
    <?prg
        local cHtml := 'Now is ' + time() + ' of ' + dtoc( date() )
        return cHtml
    ?>
<hr>
<h3>Code html from prg section...</h3>
```

Podemos poner tantos bloques de código prg como queramos.



Inyectando código HTML dentro de PRG

Y finalmente para acabar de rizar el rizo, dentro de un bloque de código prg, podemos insertar un bloque de código html, para no tener que montar sentencias html concatenando strings. Para inyectar code html dentro de un bloque prg usaremos

BLOCKS VIEW <v> [PARAMS ...]

Tenemos que pasar al bloque una variable que es donde se le asignara el código. Si deseamos también podríamos pasarle variables de dentro del prg al bloque para poderlas usar usando PARAMS <mi_var>, <mi_otra_var>,...

En el caso de pasar variables al BLOCKS VIEW, desde dentro las usaremos poniéndolas dentro de las etiquetas <\$... \$>

Imaginaros este ejemplo inyectando código prg y dentro del código prg queremos montar una salida de html

<pre><h1>Test PRG</h1> <hr> <?prg local nI local cHtml := '' for nI := 1 to 5 cHtml += '<div style="color:green;">' cHtml += 'This is loop :' + str(nI) cHtml += '</div>' next retu cHtml ?> <hr> <h3>Code html from prg section...</h3></pre>	<pre><h1>Test PRG</h1> <hr> <?prg local nI local cHtml := '' for nI := 1 to 5 BLOCKS VIEW cHtml PARAMS nI <div style="color:green;"> This is loop : <\$ nI \$> </div> ENDTEXT next retu cHtml ?> <hr> <h3>Code html from prg section...</h3></pre>
--	--

En grandes porciones de código es más limpio y fácil de codificar poniéndolo todo dentro de un BLOCKS VIEW

Y con esto cerramos el tema de las views y como se puede apreciar podemos ir desde montar un código html puro y duro, a ayudarnos insertando code prg, hasta inyectando código html, dentro de código prg. Un abanico de opciones que podremos usar.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Si creas las entradas en el router de estos dos ejemplos los podrás ejecutar fácilmente, y podrás comprobar que el sistema funciona correctamente.

Navegador → Router → View → Navegador

Quizás con lo que ya sabemos ya puedes pensar que tengas suficiente y quizás según hasta qué punto puede ser así. Puedes pensar que me creo una vista que abramos una tabla, busquemos p.e. los datos de un registro y los mostremos en la misma vista. Hagamos una prueba

Crearemos un directorio en nuestro proyecto llamado data y pondremos allá nuestra querida tabla test.dbf que estará indexada por varios campos entre ellos uno por "state"

Si os acordáis al inicio en nuestro .htaccess definimos esta variable de entorno

```
SetEnv PATH_DATA "/go/data/"
```

Ahora es el momento de usarla. Cuando usemos harbour que se ejecutará en nuestro server, nos referimos a nuestros path con toda la ruta real como lo hacemos siempre, no con la relativa. Esto quiere decir que para saber donde tenemos nuestros datos en nuestro proyecto podríamos hacer:

```
Local cPath := AP_GetEnv( "DOCUMENT_ROOT" ) + AP_GetEnv( "PATH_DATA" )
```

Esto seguramente nos apuntara si tenemos xampp instalado a c:\xampp\htdocs\go\data

Yo acostumbro en el index crearme funciones que me faciliten este path, porque las vas a usar mucho. Por ejemplo, en el index.prg añadiría esta función

```
function AppPathData()  
  
return AP_GetEnv( "DOCUMENT_ROOT" ) + AP_GetEnv( "PATH_DATA" )
```

Esta función ya es visible desde este momento desde controladores, vistas, modelos,...

Vamos a crear una view que me cree una página web y me muestre los usuarios de un estado.

```
<h1>Users was born NY - (New York)</h1>  
<hr>  
  
    <?prg  
        local nCount := 0  
        local cHtml := ''  
  
        USE ( AppPathData() + '\test.dbf' ) SHARED NEW VIA 'DBFCDX'  
        SET INDEX TO ( AppPathData() + '\test.cdx' )  
  
        cAlias := Alias()
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
OrdSetFocus( 'state' )

DbSeek( 'NY' )

cHtml += '<pre>'

while (cAlias)->state == 'NY' .and. (cAlias)->( ! Eof() )

    nCount++

    cHtml += (cAlias)->first + ' '
    cHtml += (cAlias)->last + ' '
    cHtml += (cAlias)->street + ' '
    cHtml += (cAlias)->city + '<br>'

    (cAlias)->( DbSkip() )

end

cHtml += '</pre>'
cHtml += '<hr>'
cHtml += '<b>Total: </b>' + ltrim(str(nCount))

retu cHtml

?>
```

<hr>

Crearemos una entrada en nuestro router para poderlo ejecutar

```
DEFINE ROUTE 'state' URL 'state' VIEW 'state.view' METHOD 'GET' OF oApp
```

Y simplemente ejecutamos → localhost/go/state



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

The screenshot shows a web browser window with the address bar displaying 'localhost/go/state'. The page title is 'Users was born NY - (New York)'. Below the title is a table with 16 rows of user data, organized into four columns. At the bottom of the table, it says 'Total: 16'.

Francois	Clark	14945 Philadelphia Street	Adelaide
Match	Kaltenhauser	11514 Willow Parkway	Niagra Falls
Shawn	Lanier	28299 Wertzville Road	Worcester
Ron	Patton	13438 Raffaele Place	Tempe
Roman	Wishengrad	2403 Washingtonian Blvd	Columbus
Chris	Young	27151 Avon Place	Lund
Byron	Lewis	2732 Santa Monica Blvd	Lake Worth
Carl	Chacon	17519 W 233rd Street	Mobile
Eduardo	Blackman	24023 Whipple Ave NW	Bellevue
Sharon	Newton	23635 South Hulen	Fullerton
Glenn	Dodson	30959 Howard Drive	Julian
Ronald	Cain	6915 Luanne Avenue	Chattanooga
Len	Schwartz	15632 National City Center	Topeka
Anne	Volz	13063 S. Wacker Drive	Brattleboro
George	Nelson	31786 SW Martin Downs Blvd	Martinez
Mishaël	Jonasson	28433 Great North Road	Plymouth

Total: 16

¿Que os parece? Sencillo con una simple view ponerle toda esta funcionalidad, pero nuestra arquitectura MVC no quiere seguir este patrón, sino más bien otro. Con lo fácil que lo teníamos 😞

¿Entonces donde fallamos? Fallamos en el concepto. Una view se tiene que encargar de pintar, dibujar, montar la página,... pero no de montar la lógica del programa, procesar datos, abrir tablas, realizar cálculos,... de todo ello se encarga nuestro Controller !

Controller

El Controller es el que se encarga de procesar toda nuestra lógica de programa, saber qué hacer con lo que nos pide el usuario, crear consultas, actualizaciones, cálculos, Esta es la finalidad del controller y una vez ha finalizado su cometido tiene varias opciones de las que vamos a ver unas cuantas, pero vamos a ver 2 que son las más importantes:

- Enviar los datos a una view para que esta monte la web usando además todos estos datos 😊
- Devolver al navegador en este caso los datos por ejemplo en formato json

Hay otras opciones que no se van a tratar de momento como enviar el resultado en formato xml, en un fichero texto ,... Lo más importante es entender la pieza del Controlador.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Los controladores estarán en el directorio `src/controller` y podremos añadir carpetas a ese directorio por si las necesitamos. Como comentamos en el apartado de las views uno de los objetivos del MVC es seguir una jerarquía y ordenar todo, para que luego nuestros mantenimientos sean más fáciles.

Un controlador es una clase con varios métodos, tiene un constructor y es muy sencillo de construir permitiéndonos escalar el programa poco a poco de una manera muy sencilla. Imaginemos que creamos un programa para gestionar una tienda virtual. Podríamos crear un controlador para gestionar todo lo relativo a productos, otro para el tema pedidos, stock,...

a.- Ejemplo de un controlador que envia los datos a una view

Siguiendo el ejemplo de la view con los usuarios de NY (New York) vamos a crear un controlador de usuarios que tendrá un método para consultar los usuarios de un "state". La estructura podría ser algo parecido a esto

```
CLASS Customer

    METHOD New( oController )                CONSTRUCTOR

    METHOD GetByState( oController )

ENDCLASS

//-----//

METHOD New( oController ) CLASS Customer

RETURN Self

//-----//

METHOD GetByState( oController ) CLASS Customer

RETURN nil

//-----//
```

Como podemos apreciar ya disponemos de un módulo (Customer) que será el encargado de gestionar todo lo q se refiere a nuestros clientes....

Si observáis el code veréis q todos los métodos recibirán un parámetro *oController* con una serie de propiedades que nos ayudarán en todo el proceso y que veremos durante el proceso.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Hemos comentado que el controlador una vez procesa datos puede revolver una respuesta al navegador o crear una petición para generar una view de salida. Cuando necesitemos enviar a una view usaremos el método `oController.View(<Nombre_View>)`

Lo primero que vamos a hacer es un crear un método dentro del controlador que no hará nada y llamará a una View para que nos monte una pantalla de entrada de datos y llamaremos a este método p.e. `Search ()`

En este punto vamos a ver como creamos una entrada en nuestro fichero de rutas `index.prg`.

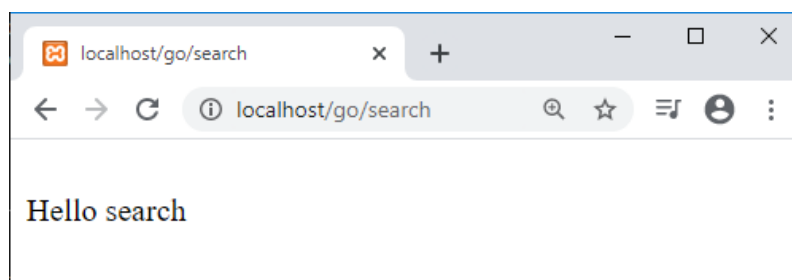
```
DEFINE ROUTE 'search' URL 'search' CONTROLLER 'search@customer.prg' METHOD 'GET' OF oApp
```

La novedad aquí es que usamos el comando `CONTROLLER` y le indicamos el nombre del controlador precedido de un `@` y el método que ha de ejecutar \rightarrow `'search@customer.prg'`

Ahora Podemos añadir el método `Search()` en el controller para ver si nos llega la petición cuando entremos por la url \rightarrow `localhost/go/search`

Para probar si funciona podemos momentáneamente poner un `? 'Hello Search'`

```
METHOD Search( oController ) CLASS Customer  
  
    ? 'Hello search'  
  
RETU nil
```



Ahora se trata de pasar el `'Hello search'` a una view, como hemos dicho con el método `oController.View()`

```
METHOD Search( oController ) CLASS Customer  
  
    oController.View( 'search.view' )  
  
RETU nil
```

Y crearemos la view en nuestra carpeta `src/view/search.view`



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
<h1>Hello Search</h1>
<hr>
```

Y con este pequeño ejemplo vemos ya un triángulo de proceso:

Navegador → Controller → View → Navegador()

Paso de parámetros Controller → View

Antes de seguir con nuestro ejemplo vamos a explicar el paso de parámetros desde un controlador a una view.

Vamos a imaginar que tenemos 2 variables (un string y un array) que hemos procesado en nuestro controller para finalmente pasarlo a una view. La sintaxis es `oController.View(<Nombre_View>, <par1>, <par2>, ...)`

```
METHOD Search( oController ) CLASS Customer

    local cDate      := DtoC( date() + 10 )
    local aFruits    := { 'Banana', 'Apple', 'Pear', 'Cherry' }

    oController.View( 'search.view', cDate, aFruits )

RETU nil
```

Y en la view vamos a montar la página recogiendo los parámetros que envía el Controller

```
<h1>Hello Search</h1>
<hr>

Now is {{ pvalue(1) }}
<br>
Now is <?prg return pvalue(1) ?>
<br>
Now is {{ PARAM 1 }}

<?prg
    local aFruits := pValue(2)
    local cHtml   := '<hr><ul>'
    local nI

    for nI := 1 to len( aFruits )
        cHtml += '<li>' + aFruits[ nI ] + '</li>'
    next
```



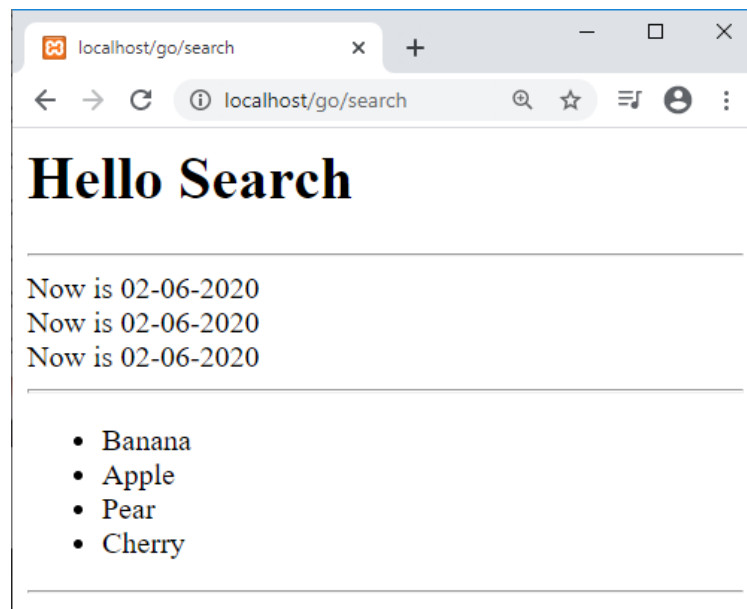
Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
cHtml += '</ul>'  
  
    retu cHtml  
?>  
  
<hr>
```

Si ejecutamos la url localhost/go/search nos parecerá la siguiente pantalla



El código de la view es para analizar y ver cómo podemos recoger los parámetros. Si observamos la fecha la podemos recoger 3 maneras

- Macrosustitución poniendo entre {{ ... }}
- Bloque de code prg entre <?prg ... ?>
- Macrosustitución usando el comando PARAM <nParametro>

El segundo bloque que muestra las frutas vemos que es un bloque prg, y recogemos el 2 parámetro (array de frutas) y luego lo procesamos como hemos visto en el apartado de las views.

Hay una cuarta manera que la comentamos a modo de información que funciona correctamente pero quizás rompe el método tradicional. Se trata desde el Controller preparar la variables que vamos a usar en la view con: App:Set(<cNameVar>, <uValue>) tantas veces como necesitemos y desde la view recoger el parámetro con App:Get(<cNameVar>)

En este punto ya debemos de entender como un controlador puede procesar sus datos y enviarlos p.e. a una view



b.- Ejemplo de un controlador que envió los datos al navegador

Pero también podemos crear un sistema que lo que queremos es que el servidor nos devuelva el resultado solo en datos, por ejemplo en formato json. Esto puede ser cuando creamos una petición de una página en Ajax y no queremos otra página nueva sino añadir datos. De esta manera podríamos crear este tipo de peticiones y añadir datos a nuestra pantalla.

Para realizarlo es muy fácil. El controller en lugar de enviar los datos a una view, lo hará al navegador. Para ello utilizará el objeto `oController:oResponse` que es el encargado de realizar las salidas con sus diferentes métodos.

oResponse

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

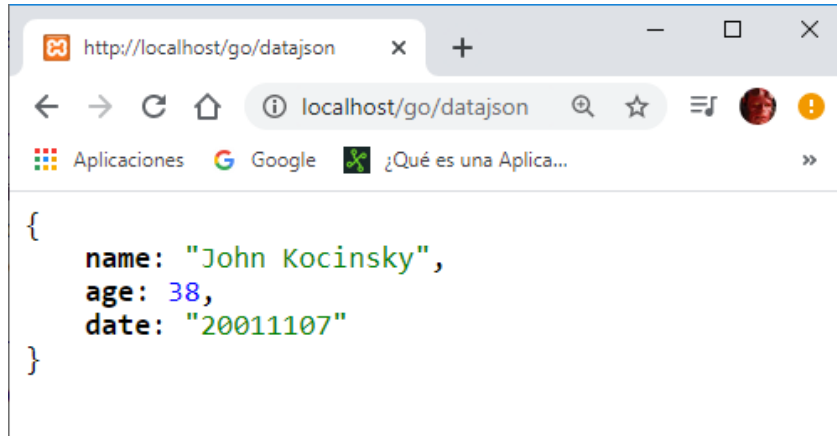
Creamos una entrada en el Router:

```
DEFINE ROUTE 'data' URL 'datajson' CONTROLLER 'data_json@customer.prg' METHOD 'GET' OF oApp
```

Y creamos el método `data_json` en el controller de esta manera.

```
METHOD data_json( oController ) CLASS Customer  
  
    local hUser := { 'name' => 'John Kocinsky', 'age' => 38, 'date' => CTod( '07/11/2001' ) }  
  
    oController:oResponse:SendJson( hUser )  
  
    RETU nil
```

Si ejecutamos → `localhost/go/datajson` nos tendría que aparecer los datos que envía nuestro controlador en formato json



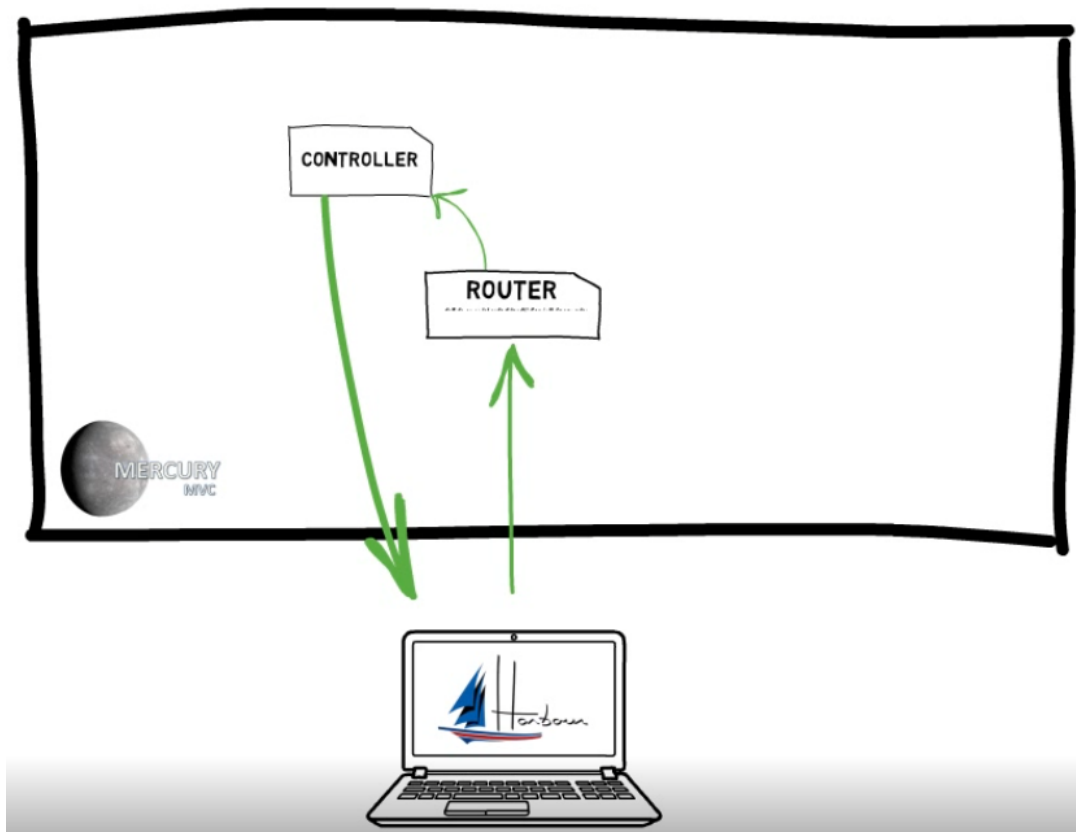
```
{
  name: "John Kocinsky",
  age: 38,
  date: "20011107"
}
```



Resumen:

- El sistema recibe una petición que se enruta a nuestro controller
- El controller procesa datos y da una respuesta en formato json

Así conceptualmente tendríamos este escenario → <https://youtu.be/VnwM4kQKKmw>





Creación de una view formulario para buscar el state

Vamos a crear en el search.view un formulario simple en html puro y duro para pedir la lista de usuarios de un state. El código html super simple podría ser algo así

```
<h1>Search Customers by State</h1>
<hr>

<form method="post">
  State: (ex. NY, IL, CO, LA,...)
  <br>
  <input type="text" name="mystate" value="NY">
  <br><br>
  <input type="submit" value="Send data">
</form>
```

Si ejecutamos localhost/go/search nos aparecerá la siguiente pantalla

localhost/go/search

Search Customers by State

State: (ex. NY, IL, CO, LA,...)

Ahora quizás viene unos de los conceptos que cuesta más de entender. Este formulario le falta añadir la "action" donde queremos que haga la petición, eso sería que programa ejecutar cuando le damos al botón. La idea es que cuando rellenemos los datos y pulsemos el botón nos ejecute nuestro método GetByState() de nuestro controlador customer. Como hacemos este paso final?



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Crearemos una ruta en nuestro index.prg que podría ser similar a este

```
DEFINE ROUTE 'getst' URL 'getbystate' CONTROLLER 'getbystate@customer.prg'  
METHOD 'POST' OF oApp
```

Deberíamos ya tener asimilado y comprender el significado de la definición del route

Ya solo nos queda poner en la view search donde lo queremos enrutar cuando pulsemos el botón. Nada tan fácil como añadir lo siguiente y que es la clave para aprender a enrutar nuestros formularios correctamente

```
<form action="{{ Route( 'getst' ) }}" method="post">
```

Lo que va hacer nuestro sistema cuando cree el formulario es sustituir `{{ Route('getst') }}` por la ruta (url) donde debe crear la petición al servidor. Si observais en el index.prg (Router) hemos definido la clave "getst" y esta apunta al controller 'getbystate@customer.prg' por lo que realmente (y si consultamos el código de la pagina una vez cargado, con botón derecho->ver código de la pagina) veremos la sustitución real

```
1 <h1>Search Customers by State</h1>  
2 <hr>  
3  
4 <form action="/go/getbystate" method="post">  
5   State: (ex. NY, IL, CO, LA,...)  
6   <br>  
7   <input type="text" name="mystate" value="NY">  
8   <br><br>  
9   <input type="submit" value="Send data">  
10 </form>
```

Es importante entender como cuando creamos nuestra view hemos de gestionar nuestras rutas. Para esto está el router y nosotros no nos debemos de preocupar por paths, urls,...

Resumen:

- Hemos creado una ruta para que nuestro sistema nos muestre un formulario para entrar datos
- La view la montamos para que el formulario apunte a /go/getbystate cuando se ejecute



Recogida de parámetros desde el Controller

Ya solo nos queda desde nuestro método `GetByState()` recoger el parámetro del "state" que se ha introducido en el formulario. Para ello el objeto `oController` que nos llega a todos nuestros métodos nos ayudará a ello con los métodos y objetos que dispone.

Objeto `oRequest`

El objeto `oController:oRequest` es el encargado en el servidor de escuchar las peticiones y recoger los parámetros. Podemos usar los siguientes métodos para recoger los parámetros

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

Básicamente usaremos el Método `Post()` y el `Get()`.

Tanto el método `Post()`, `Get()`, `Request()` tienen la siguiente definición:

```
oController:oRequest:Post( <cKey>,[<cValueDefault>], [<cFormat>] )  
oController:oRequest:Get( <cKey>,[<cValueDefault>], [<cFormat>] )  
oController:oRequest:Request( <cKey>,[<cValueDefault>], [<cFormat>] )
```

<cKey> Nombre del parámetro
<cDefault> Valor por defecto. Por defecto es una cadena vacía
<cFormat> Si queremos formatear los datos. Pueden ser los típicos de harbour: 'C', 'N', 'D'



Mercury

Modelo/Vista/Controlador

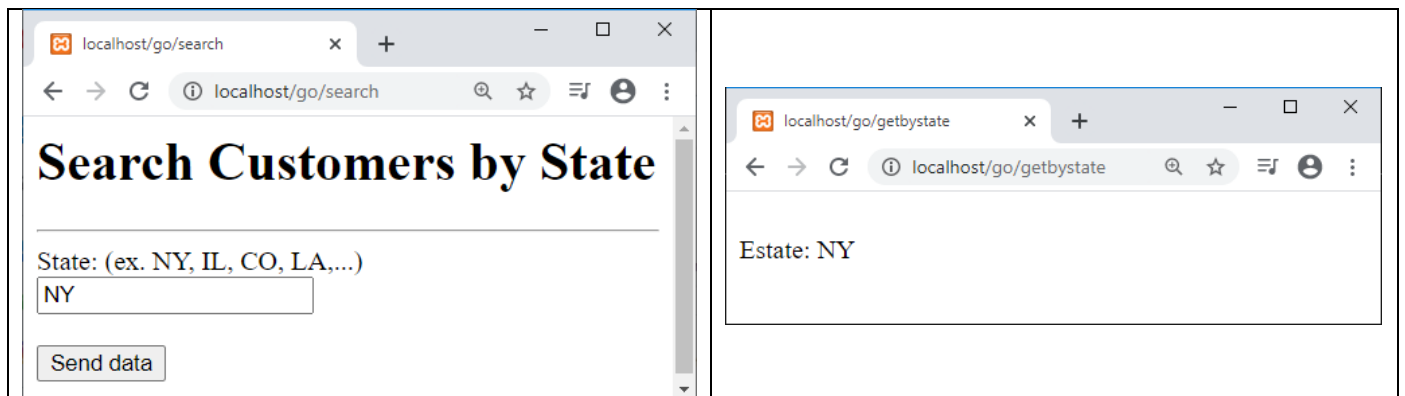
Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

En el formulario si nos fijamos, el nombre de la variable es "mystate" y sabemos que el formulario lo ejecutamos por el método "post" recuperaríamos la variable de la siguiente manera.

```
local cState := oController:oRequest:Post( 'mystate' )
```

Si hacemos la prueba y modificamos el controller para que nos muestre la variable que nos llega del formulario de esta manera podremos comprobarlo. Es una buena manera para testearlo.

```
METHOD GetByState( oController ) CLASS Customer  
  
    local cState := oController:oRequest:Post( 'mystate' )  
  
    ? 'Estate: ', cState  
  
RETU nil
```



Resumen

- Creación de una ruta "search" que nos va a un controlador que lo redirige a la view "search"
- La vista "search" monta el formulario que apunta al método getbyst del controlador cuando lo ejecute
- El método GetByState() vemos como recoge el parámetro "mystate" y lo muestra en la pantalla



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Recoger parámetro y buscar en la tabla

Ahora que sabemos cómo recuperar los parámetros ya tenemos todas las piezas del puzle. Solo tenemos que añadir el código de buscar en la tabla el "state" que hemos recibido y poner los registros en una tabla. Para saber si lo realizamos bien al final mostraremos el array

```
METHOD GetByState( oController ) CLASS Customer

    local cState := oController:oRequest:Post( 'mystate' )
    local aRows  := {}
    local nCount := 0

    USE ( AppPathData() + '\test.dbf' ) SHARED NEW VIA 'DBFCDX'
    SET INDEX TO ( AppPathData() + '\test.cdx' )

    cAlias := Alias()

    (cAlias)->( OrdSetFocus( 'state' ) )
    (cAlias)->( DbSeek( cState ) )

    while (cAlias)->state == 'NY' .and. (cAlias)->( ! Eof() )

        nCount++

        Aadd( aRows, { 'first'    => (cAlias)->first,;
                      'last'     => (cAlias)->last,;
                      'street'   => (cAlias)->street,;
                      'city'     => (cAlias)->city;
                    })

        (cAlias)->( DbSkip() )
    end

    ? aRows

RETU nil
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
localhost/go/getbystate x +
localhost/go/getbystate

[{"first":"Francois ","last":"Clark ","street":"14945 Philadelphia Street ","city":"Adelaide "}, {"first":"Match","last":"Kaltenhauser ","street":"11514 Willow Parkway ","city":"Niagra Falls "}, {"first":"Shawn ","last":"Lanier","street":"28299 Wertzville Road ","city":"Worcester "}, {"first":"Ron ","last":"Patton ","street":"13438 Raffaele Place","city":"Tempe "}, {"first":"Roman ","last":"Wishengrad ","street":"2403 Washingtonian Blvd ","city":"Columbus "}, {"first":"Chris ","last":"Young ","street":"27151 Avon Place ","city":"Lund "}, {"first":"Byron ","last":"Lewis","street":"2732 Santa Monica Blvd ","city":"Lake Worth "}, {"first":"Carl ","last":"Chacon ","street":"17519 W 233rd Street ","city":"Mobile "}, {"first":"Eduardo ","last":"Blackman ","street":"24023 Whipple Ave NW ","city":"Bellevue "}, {"first":"Sharon ","last":"Newton ","street":"23635 South Hulen ","city":"Fullerton "}, {"first":"Glenn","last":"Dodson ","street":"30959 Howard Drive ","city":"Julian "}, {"first":"Ronald ","last":"Cain ","street":"6915 Luanne Avenue ","city":"Chattanooga "}, {"first":"Len ","last":"Schwartz ","street":"15632 National City Center","city":"Topeka "}, {"first":"Anne ","last":"Volz ","street":"13063 S. Wacker Drive ","city":"Brattleboro "}, {"first":"George ","last":"Nelson ","street":"31786 SW Martin Downs Blvd ","city":"Martinez "}, {"first":"Mishael","last":"Jonasson ","street":"28433 Great North Road ","city":"Plymouth "}]
```

Vemos como el controlador, el encargado de procesar los datos ha realizado su proceso correctamente. Ahora quedaría al fin el último paso. Una vez el controlador finaliza su proceso, lo que debería hacer es pasarle a la vista los datos para que haga su trabajo: pintar la pantalla.

Pasar el resultado a la view

Una vez finalizado el controlador llamaremos a una vista que se encargue de crear la web. La llamaremos listbystate.view y el código simplon podría ser algo como esto

```
<h1>Users was born in {{ PARAM 1 }} </h1>
<hr>

<?prg
    local aRows := PValue(2)
    local nLen := len( aRows )
    local cHtml := '<pre>'
    local nI, hRow

    for nI := 1 TO nLen

        hRow := aRows[nI]

        cHtml += hRow[ 'first' ] + ' ' + hRow[ 'last' ] + ' ' + hRow[ 'street' ] + '<br>'

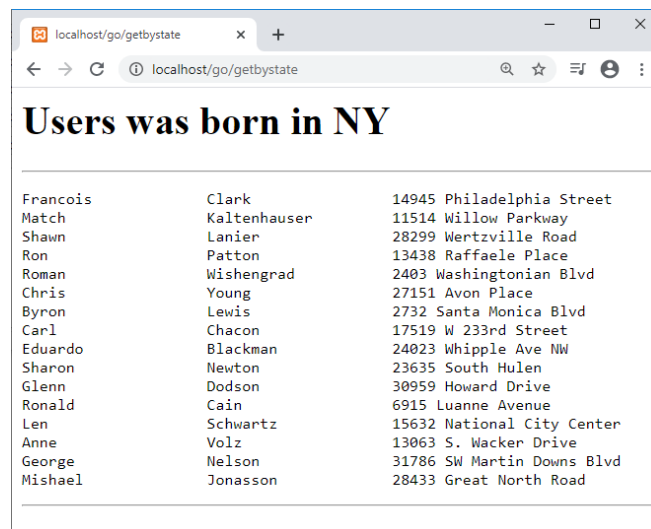
    next

    cHtml += '</pre>'

    retu cHtml
?>

<hr>
```

Y el resultado final vendría a ser una salida en pantalla similar a esta



The screenshot shows a web browser window with the address bar displaying 'localhost/go/getbystate'. The page title is 'Users was born in NY'. Below the title is a table with three columns: Name, Last Name, and Address. The table contains 18 rows of data.

Francois	Clark	14945 Philadelphia Street
Match	Kaltenhauser	11514 Willow Parkway
Shawn	Lanier	28299 Wertzville Road
Ron	Patton	13438 Raffaele Place
Roman	Wishengrad	2403 Washingtonian Blvd
Chris	Young	27151 Avon Place
Byron	Lewis	2732 Santa Monica Blvd
Carl	Chacon	17519 W 233rd Street
Eduardo	Blackman	24023 Whipple Ave NW
Sharon	Newton	23635 South Hulen
Glenn	Dodson	30959 Howard Drive
Ronald	Cain	6915 Luanne Avenue
Len	Schwartz	15632 National City Center
Anne	Volz	13063 S. Wacker Drive
George	Nelson	31786 SW Martin Downs Blvd
Mishael	Jonasson	28433 Great North Road



Resumen

- Creación de una ruta search que nos va a un controlador que lo redirije a la view Search
- La vista search mona el formulario que apunta al método getbyst del controlador cuando lo ejecute
- El método GetByState() vemos como recoge el parámetro "mystate" y abre la tabla para cargar los datos
- Lamamos a la View listbystate y le pasamos los datos para que pinte la pantalla

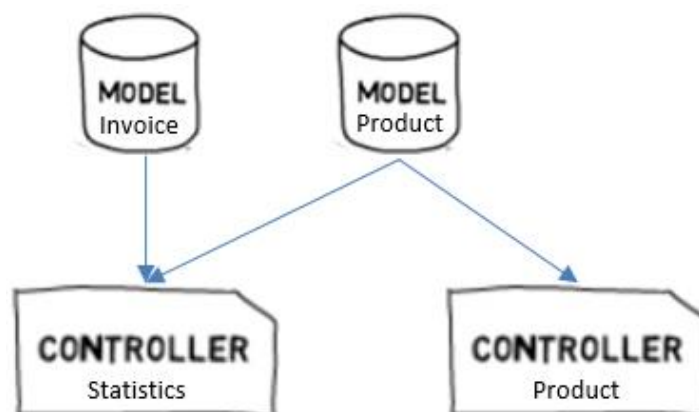
Modelo

De la misma manera que todo lo teníamos en la view inicialmente, ahora hemos extraído la parte del proceso de datos en el controlador y separamos ya un poco el proceso de datos de la view. ¿Podemos subsistir así? Pues seguramente si, como cuando lo teníamos todo en la view, pero no lo tenemos del todo correcto, aun 😞

Básicamente el modelo es la parte que conecta con nuestros datos y crea sus entradas y salidas. El objetivo principal es proveernos de los datos que puede ser una tabla dbf, como una bd de mysql, Oracle, arrays de datos definidos,...

Entonces debemos extraer lo que hemos hecho hasta ahora del controlador y encapsularlo en un modelo de datos que se encargara de tratar la tabla Customer. Al final el controlador si necesita ese modelo o modelos los abrirá pedirá los datos que necesite para luego procesarlos.

A partir de aquí hemos de pensar que nuestra aplicación a medida que crezca tendrá varios controllers como hemos comentado anteriormente y cada controller usar su/s modelos de datos, pero pueden haber controllers que usen un mismo modelo, por lo que, si creamos así nuestro sistema, nuestro modelo es universal para cualquier controlador que lo necesite



Creando el Modelo Customer

Para nuestro ejercicio crearemos un modelo sencillo de datos que simplemente abrirá la tabla de datos y tendrá un método que se llamara RowsByState(cState). Podríamos hacerlo sencillo de esta manera

```

CLASS CustomerModel

    DATA cAlias

    METHOD New() CONSTRUCTOR

    METHOD RowsByState( cState )

ENDCLASS

//-----//

METHOD New() CLASS CustomerModel

    USE ( AppPathData() + 'test.dbf' ) SHARED NEW VIA 'DBFCDX'
  
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
SET INDEX TO 'test.cdx'

::cAlias := Alias()

RETU SELF

// -----

METHOD RowsByState( cState ) CLASS CustomerModel

    local aRows := {}

    DEFAULT cState TO ''

    (::cAlias)->( OrdSetFocus( 'state' ) )
    (::cAlias)->( DbSeek( cState ) )

    while (::cAlias)->state == cState .and. (::cAlias)->( ! Eof() )

        Aadd( aRows , {      'first'      => (::cAlias)->first,;
                           'last'       => (::cAlias)->last,;
                           'street'    => (::cAlias)->street,;
                           'city'      => (::cAlias)->city,;
                           'zip'       => (::cAlias)->zip,;
                           'salary'    => (::cAlias)->salary ;
                        })

        (::cAlias)->( DbSkip() )

    end

RETU aRows

// -----
```

Una vez tenemos ya el modelo creado el cambio a realizar en el Controller serian 2:

```
METHOD GetByState( oController ) CLASS Customer

    local cState := oController:oRequest:Post( 'mystate' )
    local oCusto := CustomerModel():New()
    local aRows := oCusto:RowsByState( cState )

    oController:View( 'listbystate.view', cState, aRows )

RETU nil
```

Y al final del contoller insertar el modelo

```
{% LoadFile( "/src/model/customer.prg" ) %}
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Hemos de pensar que nosotros cuando creamos una aplicación web, a diferencia de una aplicación p.e. Windows en que nosotros compilábamos y luego enlazábamos librerías que necesitábamos en el programa, aquí en la web indicaremos de esta manera si necesitamos algún fichero y se resolverá en tiempo de ejecución.

Y esto es todo ¡!!! 😊

Con esta parte debemos pensar que básicamente abstraemos todo el modelo de datos, que compartimos modelo de datos y... si algún día cambiamos el modelo Customer.prg por otro pero que en lugar de acceso a dbfs lo hace a mysql, SOLO hemos de cambiar uno por otro sin necesidad de tocar nada mas de nuestra aplicación !



Resumen de los procesos que intervienen en esta consulta:

- 1- Hemos creado una ruta en nuestro index.prg para gestionar "search"

```
DEFINE ROUTE 'search' URL 'search' CONTROLLER 'search@customer.prg'  
METHOD 'GET' OF oApp
```

- 2- Cuando llega la petición al controller la redirige a la view "search.view", que crea el formulario para enviar al navegador
- 3- Este formulario que llegara al navegador tiene en la etiqueta <form> la ruta que deberá tomar cuando pulsemos el botón de enviar datos.

```
<form action="{ Route( 'getst' ) }" method="post">
```

- 4- Esto implica que debemos tener en nuestro router (index.prg) creada una ruta para "getst" que la acepte para método 'post' que es la que usaremos en el formulario

```
DEFINE ROUTE 'getst' URL 'getbystate' CONTROLLER 'getbystate@customer.prg'  
METHOD 'POST' OF oApp
```

- 5- también tendrá la variable que se enviara al servidor "mystate"

```
<input type="text" name="mystate" value="NY">
```

- 6- En el controller customer tendremos un método 'getbystate' que será el que escuche la petición
- 7- El método GetByState() recuperará los parámetros del formulario, en este caso 'mystate'

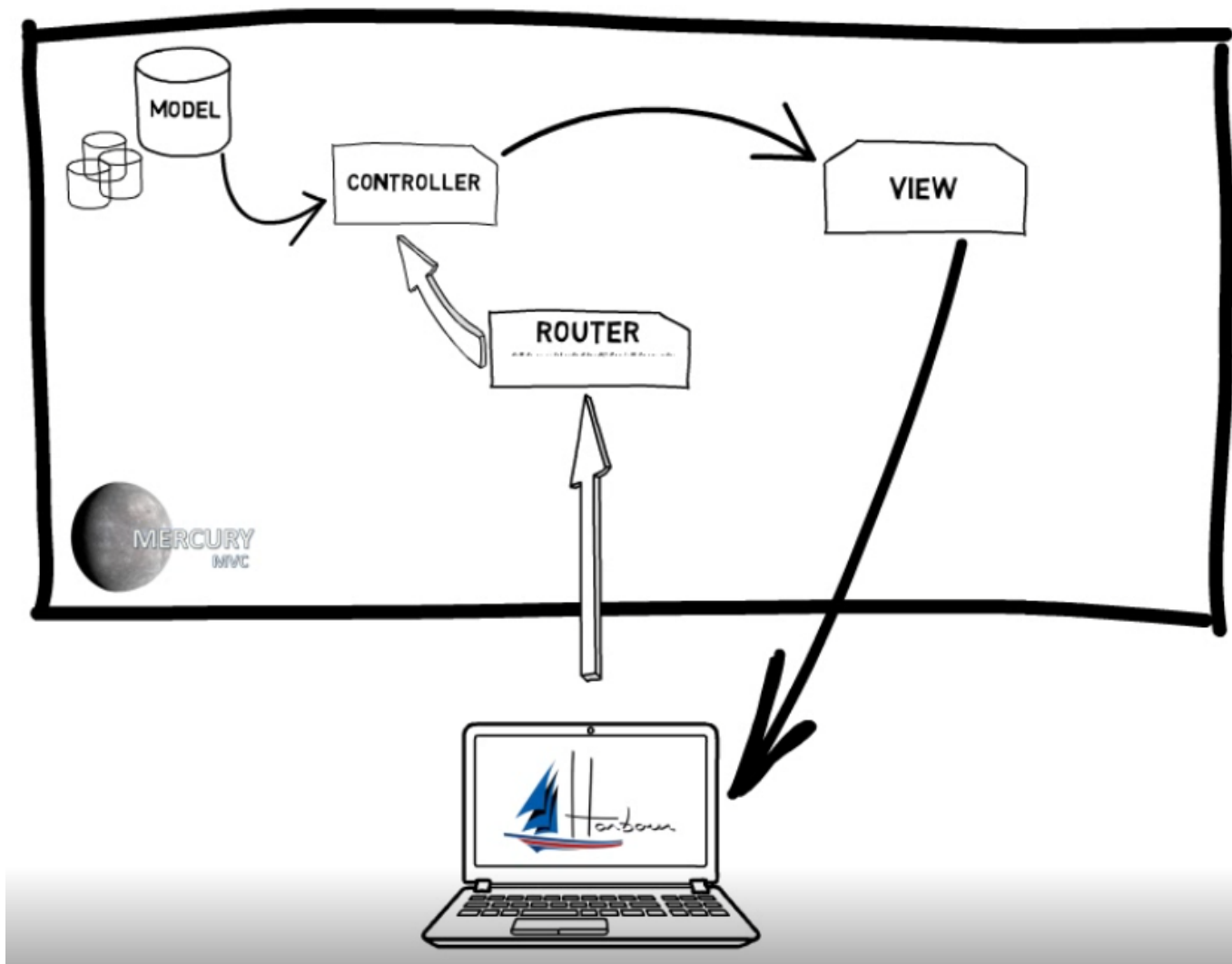
```
local cState := oController:oRequest:Post( 'mystate' )
```

- 8- Abriremos una instancia del modelo de datos Customer y le pediremos los datos del "state" que hemos recuperado

```
local oCusto := CustomerModel():New()  
local aRows  := oCusto:RowsByState( cState )
```

- 9- Enviamos los datos a la vista "listbystate.view" para que los pinte. Esta view recuperará los datos y los pintará

Este es un circuito típico del MVC y si hemos llegado hasta aquí, nuestra mente ya debe ser capaz de ver este esquema → <https://youtu.be/FN9Urv0Ouxs>





Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Y si seguimos analizando el esquema veremos que hemos dado 2 vueltas:

- 1- Cuando solicitamos el "search" que es la pantalla para entrar datos
- 2- Cuando entramos el dato y solicitamos la información

Esto es la base de MVC. Si conseguimos entenderla y aplicarla nos dará una gran seguridad y productividad con el tiempo.

A parte de las carpetas de datos, lib, la lógica de la aplicación que hemos creado reside en estas carpetas y ficheros.

```
/go
  /src
    /controller
      customer.prg
    /model
      customer.prg
    /view
      search.view
      listbystate.prg

/index.prg
```

Sabemos rápidamente que tocar y donde ir si tenemos un problema en la pantalla, o en el acceso de datos, o su proceso... Realmente es un sistema muy ordenado y potente que nos ayudará en el diseño de nuestras aplicaciones



2ª Parte - Avanzando en el patrón

Si hemos conseguido llegar hasta aquí tenemos la parte más importante lograda y es la concepción de todo el sistema. Entender por donde entra, sale, que hace cada módulo y que función tienen.

Pero el tema no se acaba aquí 😞 , podemos hacer mucho más para lograr que nuestro sistema sea eficaz, ágil, robusto,...

La primera ley en un backend es SIEMPRE validar los datos.

Mercury dispone de una clase TValidator() que te puede ayudar en esa labor, no es obligatoria y cada uno puede usar su sistema ya que se puede controlar de muchas maneras.

Uno de los errores más comunes es usar pluggins de javascript para validar entradas de datos y esto queda muy bien en la pantalla cuando entramos datos, pero siempre hemos de pensar en cómo se puede alterar el sistema porque siempre hay alguien...que lo intenta.

Siguiendo el ejemplo de la búsqueda de customer de un "state" lo primero que hemos de pensar es que, al recibir el parámetro, sabemos que el formato que ha de tener el siguiente formato:

- Caracteres
- Longitud de 2
- Mayúsculas

No hay mas, porque lo hemos definido así. Pues esto es lo que debemos dejar pasar y si no hemos de dar un mensaje de error.

TValidator

Esta clase optativa es muy fácil de usar y nos servirá en la mayoría de los casos para poder controlar estos supuestos

- 1- Crearemos un objeto TValidator

```
local oValidator := TValidator():New()
```

- 2- Crearemos unas reglas de validación que se basa en un hash de las variables que queremos validar. Cada clave tendrá un string con reglas, separadas por el carácter | para separar las diferentes reglas, p.e.

```
local hRoles := { 'mystate' => 'required|string|len:2' }
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Donde especificamos que la variable "mystate" ha de tener las siguientes validaciones:

- Required – obligatorio, ha de existir el parámetro
- Ser de tipo string, no solo numérico
- Longitud exacta de 2 caracteres

3- Pondremos en un hash las variables que queramos validar, en este caso

```
hData[ 'mystate' ] := oController:oRequest:Post( 'mystate' )
```

4- Validamos las reglas → oValidator:Run(hRoles)

5- Si el método nos da falso (.F.), podemos recuperar el error con oValidator:ErrorString()

6- Enviar el error a una view que puedes ser o no la misma, p.e

```
oController:View( 'search.view', oValidator:ErrorString() )
```

La view ya habrá de gestionar si recibe o no el 1 parámetro y en caso de recibirlo mostrarlo.

7- En el caso de que le validator lo pase, seguiremos el flujo como hasta ahora...

El objeto TValidator() tiene un método para "formatear" la variable. Esto quiere decir que podemos pasar la validación de tipo de variable, pero quizás necesitemos formatear la variable, p.e. sabemos que nuestro campo "state" ha de estar en mayúsculas. En este caso podemos usar el método formatter() al que le pasamos un hash con las variables y su tipo formato, p.e.

```
local hFormat := { 'mystate' => 'upper' }
```

```
oValidator:Formatter( hData, hFormat )
```

En esta tabla se muestran las distintas validaciones que podemos controlar

Rol	Descripción
required	Campo obligatorio. Es necesario recibirlo
numeric	Campo debe estar formado por dígitos. Debemos pensar que siempre recibimos los valores en formato string, pero aquí especificamos que deseamos que sean números, independientemente que después los podamos convertir de string a numero
string	Campo formado por caracteres y/o números
len:nnn	Longitud del campo ha de ser igual a nnn
max:nnn	Máxima longitud de campo si lo hemos cambiado a formato numérico ha de ser nnn
maxlen:nnn	Máxima longitud del campo string
minlen:nnn	Mínima longitud del campo string



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Vemos como quedaría el controlador:

```
METHOD GetByState( oController ) CLASS Customer

local oCusto      := CustomerModel():New()
local oValidator  := TValidator():New()
local hRoles      := { 'mystate' => 'required|string|len:2' }
local hFormat     := { 'mystate' => 'upper' }
local hData       := {=>}
local aRows

//  Recupero Datos -----

hData[ 'mystate' ] = oController:oRequest:Post( 'mystate' )

//  Valido datos -----

if ! oValidator:Run( hRoles )

    oController:View( 'search.view', oValidator:ErrorString() )

    return nil

endif

oValidator:Formatter( hData, hFormat )

// -----

aRows := oCusto:RowsByState( hData[ 'mystate' ] )

oController:View( 'listbystate.view', hData[ 'mystate' ], aRows )

RETU nil
```

Para finalizar observad en que si pon pasa la validación, llamamos en este caso a la misma view y le pasamos el error. La view habrá de detectar si se le pasa este parámetro y mostrarlo.

```
<h1>Search Customers by State</h1>
<hr>

<?prg
    local cError := pvalue(1)

    if valtype( cError ) == 'C'
        return '<b>Error: </b>' + cError + '<hr>'
    endif

    return ''

?>

<form action="{{ Route( 'getst' ) }}" method="post">
    State: (ex. NY, IL, CO, LA,...)
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

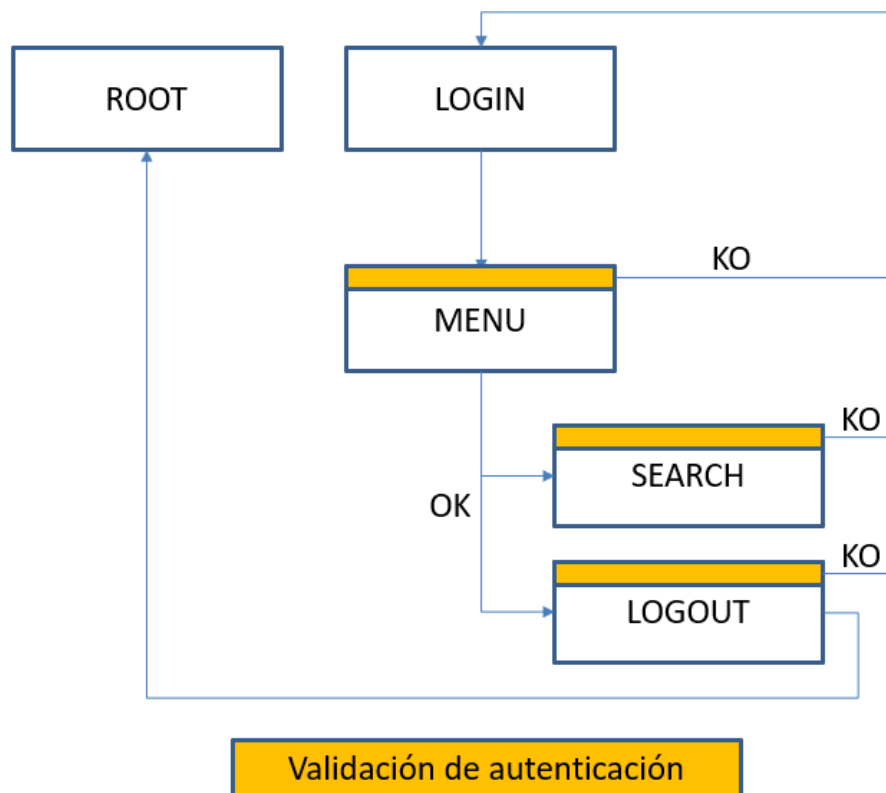
```
<br>  
<input type="text" name="mystate" value="NY">  
<br><br>  
<input type="submit" value="Send data">  
</form>
```

Y esto es todo en este apartado. Aquí lo importante es entender que debemos validar siempre las entradas a nuestro servidor. Si ejecutamos este ejemplo y entramos por ejemplo el valor "z1aBc", la aplicación nos redirigirá de nuevo a la view y nos mostrará el error.



Autenticación

Uno de los temas más importantes a tener en cuenta es controlar los accesos a nuestro sistema. Seguramente habrán módulos de nuestra aplicación que si no estas autenticado no debes de tener acceso. Este capítulo va sobre la autenticación y aprovechando lo que hemos hecho vamos a desarrollar este sistema



Vamos a aprender como nuestros distintos controllers pueden ser protegidos contra acceso si un usuario no está autenticado. Haremos el ejercicio lo más simple para asimilar fácilmente el proceso. En el diagrama vemos un rectángulo amarillo que representa el middleware, la lógica que comprueba si se está autenticado o no.

No voy a comentar las pantallas que sean prácticamente una copia de ejemplos que hayamos visto, solo las novedades para crear el sistema.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Login

El sistema login será solo un formulario, una view. No necesitamos más.

Entrada en router

```
DEFINE ROUTE 'login' URL 'login' VIEW 'login.view' METHOD 'GET' OF oApp
```

View: login.view (basado en el formulario search.view)

```
<h1>Autenticacion</h1>
<hr>

<?prg
  local cError := pvalue(1)

  if valtype( cError ) == 'C'
    return '<b>Error: </b>' + cError + '<hr>'
  endif

  return ''
?>

<form action="{{ Route( 'auth' ) }}" method="post">
  User <input type="text" name="user" value="demo">
  Password <input type="password" name="psw" value="">
  <br><br>
  <input type="submit" value="Login">
</form>
```

Vemos que el formulario lo "enrutamos" a "auth" via post

```
DEFINE ROUTE 'auth' URL 'auth' CONTROLLER 'auth@access.prg' METHOD 'POST' OF oApp
```



Controller de Validacion

Crearemos el controller access.prg que usaremos para validar y hacer logout

Auth()

El objetivo del método es autenticar los datos entrados en el formulario login. El proceso será el siguiente:

- Recuperamos datos
- Validamos Datos
- Comprobamos identidad (user/psw)
- Si KO enviamos de nuevo a Login
- Si OK creamos validación en nuestro sistema y redirigimos a menú

```
METHOD Auth( oController ) CLASS Access

local oValidator := TValidator():New()
local hRoles     := { 'user' => 'required|string|maxlen:8|minlen:1' , 'psw' => 'required' }
local hFormat    := { 'user' => 'lower' }
local hData      := {}
local hToken     := {}
local aRows

//  Recupero Datos -----

hData[ 'user' ] := oController:oRequest:Post( 'user' )
hData[ 'psw' ]  := oController:oRequest:Post( 'psw' )

//  Valido datos -----

if ! oValidator:Run( hRoles )

    oController:View( 'login.view', oValidator:ErrorString() )

    return nil

endif

oValidator:Formatter( hData, hFormat )

//  Comprobar identidad -----

if hData[ 'user' ] == 'demo' .and. hData[ 'psw' ] == '1234'

    //  Entro datos opcionales, que los puedo consultar cada vez que entren
    //  No se ha de poner informacion sensible...

    hToken := { 'in' => time(), 'user' => hData[ 'user' ] }
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
//      Inicamos nuestro sistema de Validación del sistema basado en JWT

oController:oMiddleware:SetAuthenticationJWT( hToken )

//      Redirijimos a pantalla principal

oController:Redirect( Route( 'welcome' ) )

else

oController:View( 'login.view', 'No es posible Auteticar. Vuelva a probarlo' )
return nil

endif

RETURN nil
```

Comentaremos a partir de la comprobación de identidad. Como se puede ver lo hacemos comparando el string y listos, pero cuando practiquéis os podéis crear un modelo de usuario para buscar y comprobar la existencia y el psw.

La idea es que si la validación es OK crearemos un TOKEN q identificara al usuario cada vez que haga una petición al server. Podemos añadir datos adicionales nuestros al token que podemos recuperar cada vez que nos hagan una petición y es lo que hacemos en el ejemplo creamos un has con un par de datos y posteriormente generamos el token con:

```
oController:oMiddleware:SetAuthenticationJWT( hToken )
```

Seguidamente redireccionaremos el sistema al menú

```
oController:Redirect( Route( 'welcome' ) )
```

Como podéis observar, enrutamos hacia "welcome" por lo que deberemos hacer una entrada en el router. Welcome será nuestro menú.

Logout()

Será el segundo método del controller access.prg y que servirá para dar de baja el token que identifica al usuario cuando cierra el sistema. Las acciones que se deben hacer son:

- Cerrar el token
- Redirigir a la página inicial ('root')



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
METHOD Logout( oController ) CLASS Access

LOCAL oMiddleware := oController:oMiddleware

oMiddleware:CloseJWT()

oController:Redirect( Route( 'root' ) )

RETU nil
```

Para realizar esta acción lo haremos con el objeto oMiddleware y su método CloseJWT() y listos. Hasta aquí ya tenemos el control de acceso a nuestra aplicación, no es complicado.

Falta (opcionalmente) un último caso en el tema de validación. Mercury por defecto asigna un nombre de cookie al programa junto con un password que encripta el token. Si queréis personalizar vuestra cookie y password, en el fichero index.prg cuando se define la App se ha de especificar de esta manera:

```
// Define App

DEFINE APP oApp TITLE 'My web application...' ;
ON INIT Config() ;
CREDENTIALS 'HshoP!2020v1' COOKIE 'MyApp'
```

Vamos a crear un menú y nos basaremos en el ejemplo del inicio de una view que saltava a otra usando el tag <a>

Welcome

```
<h1>Bienvenido a nuestro sistema</h1>
<hr>
Menu
<br>
<ul>
<li><a href="{{ Route( 'search' ) }}">Customers by State</a></li>
<li><a href="{{ Route( 'logout' ) }}">Logout</a></li>
</ul>
```

Código sencillo, solo 2 links a:

- Search (que es el ejemplo que hemos hecho anteriormente de listado por state)
- Logout (que se usa para desconectar la autenticación)





Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

En el router definimos welcome

```
DEFINE ROUTE 'welcome' URL 'welcome' CONTROLLER 'welcome@menu.prg' METHOD 'GET' OF oApp
```

Recordad que la ruta "welcome" la usamos en la autenticación

El método logout ya lo tenemos definido en el controller Access, por lo que solo nos queda darlo de alta en el router para que lo podamos enrutar desde welcome

```
DEFINE ROUTE 'logout' URL 'logout' CONTROLLER 'logout@access.prg' METHOD 'GET' OF oApp
```

Middleware

Y el punto final: Proteger aquellos módulos que queremos que no se acceda si no se esta autenticado, en este caso queremos prohibir el acceso al welcome y al search . Lo que se tiene que hacer es insertar en cada controller en el método new lo siguiente:

```
#include {% MercuryInclude( 'lib/mercury' ) %}  
  
CLASS Menu  
  
    METHOD New( oController )                                CONSTRUCTOR  
  
    METHOD Welcome( oController )  
  
ENDCLASS  
  
//-----//  
  
METHOD New( oController ) CLASS Menu  
  
    AUTENTICATE CONTROLLER oController ERROR VIEW 'login'  
  
    RETU Self  
  
//-----//  
  
METHOD Welcome( oController ) CLASS Menu  
  
    oController:View( 'welcome.view' )  
  
    RETU nil  
  
//-----//
```

Que hace realmente este comando? Cuando se intenta acceder al controlador verifica que este autenticado y si no lo esta lo dirigimos a la pantalla de login



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Siempre que usemos comando preprocesados como estos, recordad de poner al inicio del programa el include de Mercury

```
#include {% MercuryInclude( 'lib/mercury' ) %}
```

A modo de curiosidad, cuando accedáis al login y os autentiqueis, si miras en el inspector, en el apartado aplicación, las cookies veréis como se os ha creado la cookie que llevara el token de seguridad. Si la borráis o hacéis un logout ya no podréis acceder a los módulos protegidos y deberéis autenticaros de nuevo

Application

Manifest

Service Workers

Clear storage

Storage

Local Storage

Session Storage

IndexedDB

Web SQL

Cookies

http://localhost

Filter

Only

Name	Value	Domain	P..	Expires / Max-...	Size	
MyApp	eyJ0eXAi...	localhost	/	2020-05-26T0...	233	

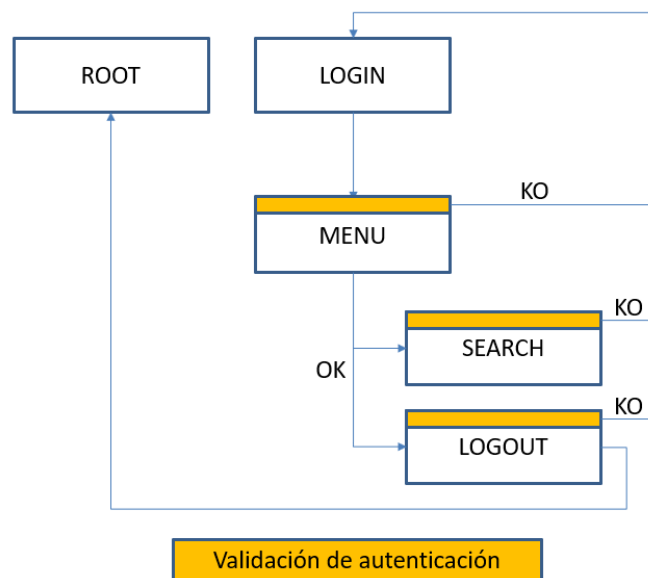
Y con este punto acabamos el ejercicio práctico. Ahora ya solo os queda escalar vuestras necesidades de manera fácil y ordenada 😊

Resumen final de nuestro sistema

Ya tenemos acabado nuestro primer sistema.

- Pantalla inicial → localhost/go/
- Sistema de autenticación → localhost/go/login
- Menú con opciones → localhost/welcome
- Modulo search → localhost/go/search
- Módulos con sistema de validación de autenticación

Observad que si intentamos acceder al módulo search o welcome sin estar autenticados, el sistema nos redirigirá automáticamente a la pantalla de login. Quizás ahora se vea y entienda más nuestro planteamiento inicial



Hecho uno, hechos todos !!! y de esta manera podremos proteger nuestros módulos de accesos indeseados.



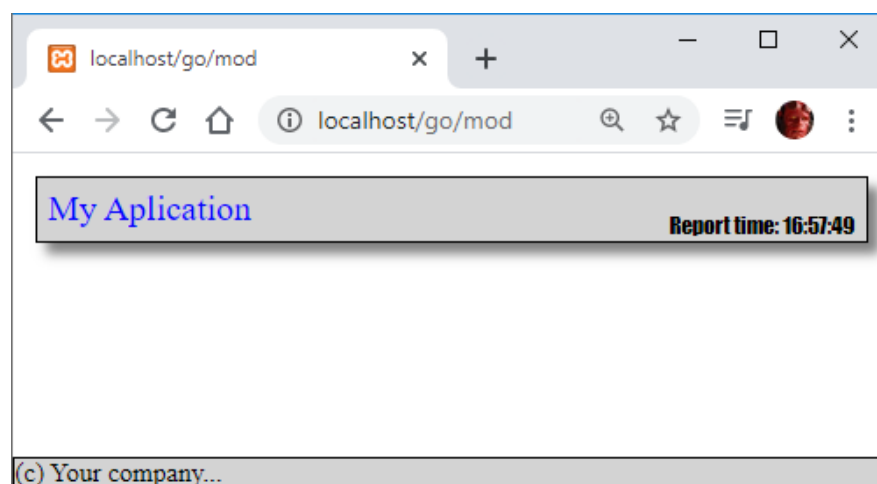
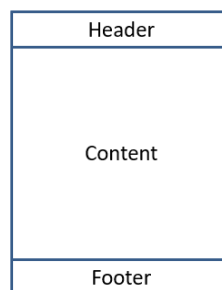
3ª Parte – Técnicas avanzadas

El motivo de esta 3ª parte es la de consolidar el sistema y añadir una serie de funcionalidades que nos facilitaran la vida a la hora de programar nuestra aplicación.

View() - View de View

Hemos visto a nivel muy conceptual como crear una vista, hiper sencilla, pero la parte del pintado de pantallas es una de las que mas dolores de cabeza nos da. Aquí también hemos de contar con la virtud del MVC que nos ayudará a optimizar code.

Imaginemos la típica página que tiene su cabecera, cuerpo y pie de página (header,content,footer). Prácticamente la cabecera y el pie de la página serán siempre los mismos o la mayoría de veces así será.





Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Lo que se trata es de dividir en diversos ficheros y usarlos cuando se necesiten. Vamos a montar un ejemplo creando un fichero que pintara la cabecera y otro el pide de pagina

myheader.view

```
<style>
body {
  margin:0px;
}

#title {
  margin: 10px;
  border: 1px solid black;
  padding: 5px;
  right: 50px;
  color: blue;
  background-color: lightgray;
  box-shadow: 5px 5px 5px grey;
}

#now {
  float: right;
  font-size: 10px;
  margin-top: 10px;
  color: black;
  font-family: fantasy;
}

</style>

<div id="title">
  My Aplication
  <div id="now">
    Report time: {{ time() }}
  </div>
</div>
```

myfooter.view

```
<style>

#copyright {

  border: 1px solid black;
  background-color: lightgray;
  bottom: 0px;
  position: absolute;
  width: 100%;
  box-sizing: border-box;
  font-size: 12px;
}

</style>
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
<div id="copyright">
  (c) Your company...
</div>
```

La idea es que desde cualquier view podamos cargar esta cabecera y este pie usando la función View(<cView>) usando {{ ... }}

```
{{ View( "myheader.view" ) }}
```

El código se simplificaría luego a la hora de crear una página de esta manera

mod-a.view

```
{{ View( 'myheader.view' ) }}

<style>

.content {
  padding: 20px;
}

</style>

<div class="content">
  <h3>Module: Mod-A</h3>
  <p>Lorem Ipsum es simplemente el texto de relleno de las imprentas y archivos de texto. Lorem Ipsum ha sido el texto de relleno estándar de las industrias desde el año 1500, cuando un impresor (N. del T. persona que se dedica a la imprenta) desconocido usó una galería de textos y los mezcló de tal manera que logró hacer un libro de textos especimen. No sólo sobrevivió 500 años, sino que tambien ingresó como texto de relleno en documentos electrónicos, quedando esencialmente igual al original</p>
</div>

{{ View( 'myfooter.view' ) }}
```

mod-b.view

```
{{ View( 'myheader.view' ) }}

<style>

.content {
  padding: 20px;
}

</style>

<div class="content">

  <h3>Module: Mod-B</h3>
```



Mercury

Modelo/Vista/Controlador

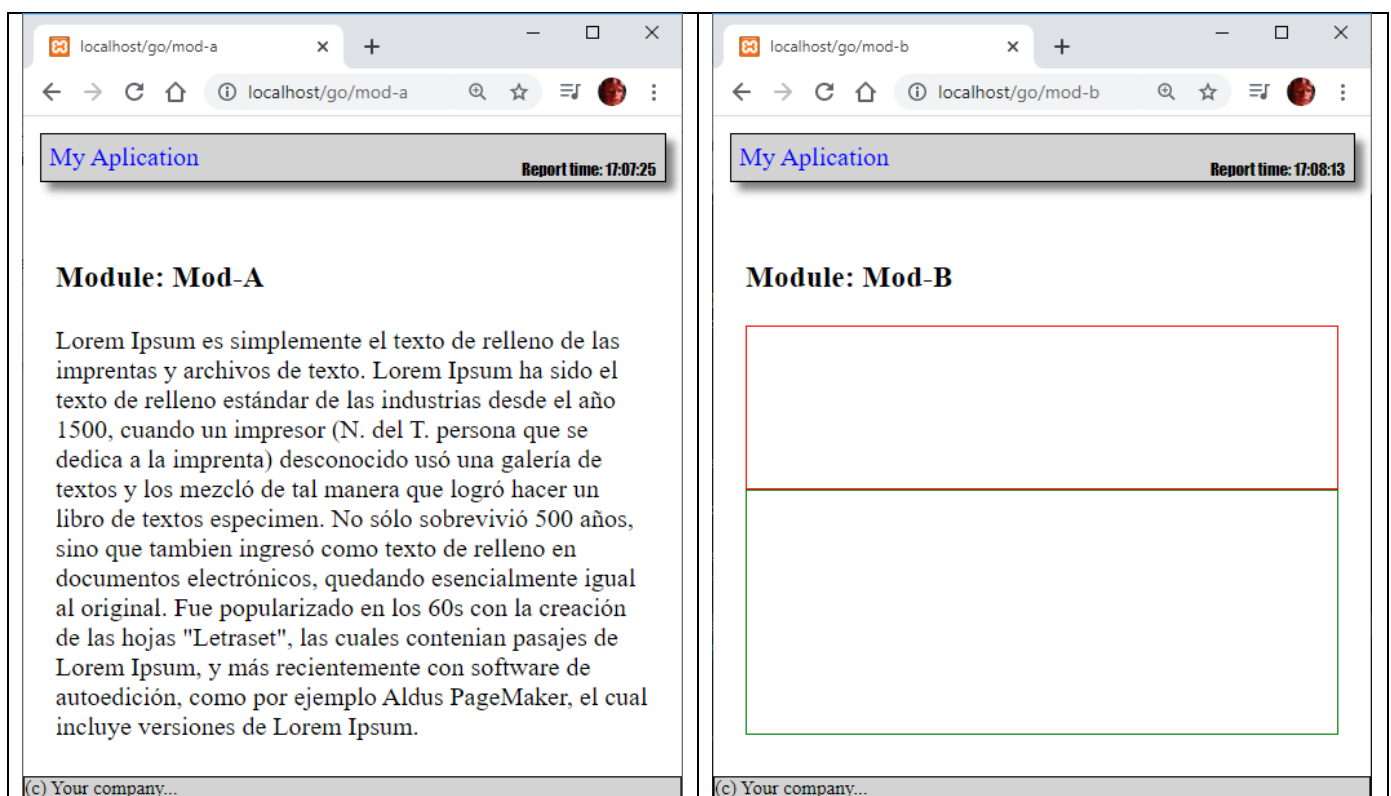
Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
<div style="border:1px solid red; height:100px">
</div>

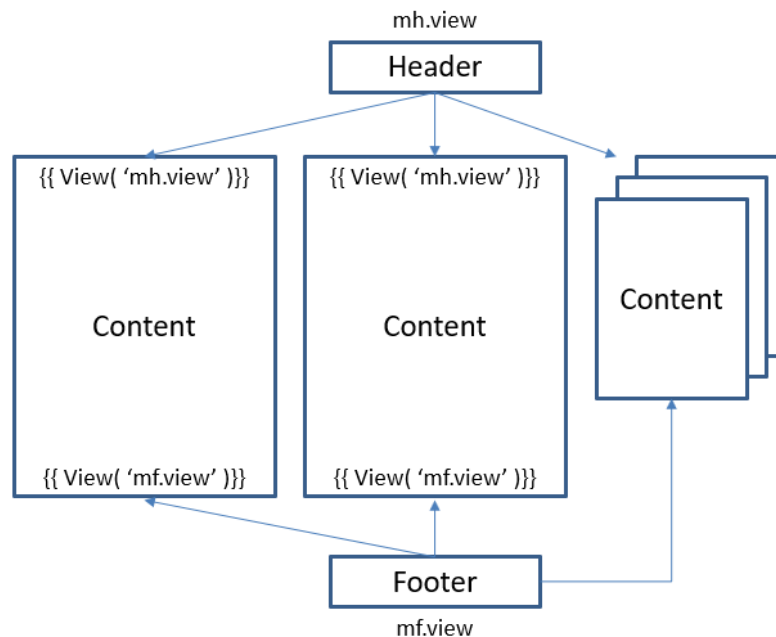
<div style="border:1px solid green; height:200px">
</div>

</div>

{{ View( 'myfooter.view' ) }}
```



De esta manera el resultado se puede apreciar que es el mismo con la diferencia que el mantenimiento es diferente. Imaginemos que tenemos 10 pantallas diseñadas y cada una con todo el código. Si queremos modificar la cabecera para todas, tendremos de modificar una por una todas las view, mientras que de esta manera modificando solo la cabecera, ya automáticamente se reflejará en todas las views donde se use.



Css()

Y para finalizar la optimización quedaría hablar del css. Cada vez más usaremos el css porque lo iremos asimilando y veremos sus bondades a la hora de maquetar, pero...seguimos poniendo el código css dentro de la propia view ? Las buenas praxis aconsejan tenerlo en un fichero separado y nosotros vamos a seguir el consejo...

Crearemos en nuestro proyecto una carpeta que se llame css y dentro crearemos en nuestro caso un fichero que llamaremos app.css . En el por ejemplo pondremos todo el css que hemos usado en estos últimos ejemplos, por ejemplo si vemos el ejemplo myhead.view podríamos poner en el app.css todo sus estilos usados quedando así

app.css

```

/*    Header */

body {
  margin:0px;
}

#title {
  margin: 10px;
  border: 1px solid black;
  padding: 5px;
  right: 50px;
  color: blue;
}
  
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
background-color: lightgray;
box-shadow: 5px 5px 5px grey;
}

#now {
  float: right;
  font-size: 10px;
  margin-top: 10px;
  color: black;
  font-family: fantasy;
}

/*      Footer */

#copyright {
  border: 1px solid black;
  background-color: lightgray;
  bottom: 0px;
  position: absolute;
  width: 100%;
  box-sizing: border-box;
  font-size: 12px;
}

/*      Content      */

.content {
  padding: 20px;
}
```

Y solo queda insertar este fichero de estilos allá donde lo podamos usar usando la función `Css()`

```
{{ Css( "app.css" ) }}
```

Para finalizar el código de myheader y myfooter quedaría de esta manera

myheader.view

```
<div id="title">
  My Aplication
  <div id="now">
    Report time: {{ time() }}
  </div>
</div>
```

myfooter.view

```
<div id="copyright">
  (c) Your company...
</div>
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Y finalmente en la página que montamos, cargaremos el css y las view de cabecera y pie de página, con lo que el código quedara muy optimizado

mod-b.view

```
{{ Css( 'app.css' ) }}  
{{ View( 'myheader.view' ) }}  
  
<div class="content">  
  
  <h3>Module: Mod-B</h3>  
  
  <div style="border:1px solid red; height:100px">  
  </div>  
  
  <div style="border:1px solid green; height:150px">  
  </div>  
  
</div>  
  
{{ View( 'myfooter.view' ) }}
```



Maquetado

Una vez terminado todo el proceso para conocer cómo funciona MVC, vamos a ver como encaja el maquetado en nuestros proyectos. Lo que hemos visto hasta ahora es la lógica de toda nuestra aplicación como encajan todas las piezas: Router, controller, model, view, middleware, validator,...

El maquetamiento es el proceso en que dejamos nuestra aplicación “bonita” 😊. No es le objetivo de este manual enseñar como funciona, ya existen cientos de manuales en la red que podéis ver sus fundamentos, pero si que representa en este punto un pequeño cambio en alguna pantalla.

Elegiremos la pantalla de Login para “maquetarla” usando Bootstrap para entender este proceso final.

La view original era esta

```
<h1>Autenticacion</h1>
<hr>

<?prg
    local cError := pvalue(1)

    if valtype( cError ) == 'C'
        return '<b>Error: </b>' + cError + '<hr>'
    endif

    return ''
?>

<form action="{{ Route( 'auth' ) }}" method="post">
    User <input type="text" name="user" value="demo">
    Password <input type="password" name="psw" value="">
    <br><br>
    <input type="submit" value="Login">
</form>
```

Para el cambio de look, crearemos una view que cargará las librerías de Bootstrap necesarias que llamaremos head.view

```
<!DOCTYPE html>
<html>

<head>

    <title>{{ App():cTitle }}</title>

    <meta charset="utf-8">

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



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
<link rel="shortcut icon" type="image/png" href="{{ AppUrlImg() + 'favicon.ico' }}" />

<!-- Bootstrap CSS CDN -->
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css">

<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.js"></script>

<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js"></script>

<script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js"></script>

<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.8.2/css/all.css">
</head>
```

Y cogiendo como base login.view la modifico quedando de esta manera y todo lo que se usa se ha visto a lo largo del manual:

- View()
- Css()
- Route()

He añadido un par de imágenes que las he puesto en la carpeta /images que formará parte de nuestro proyecto

```
{{ View( 'head.view' ) }}
{{ Css( 'login.css' ) }}

<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  
  <a class="navbar-brand" href="{{ Route( 'root' ) }}">&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&Go !</a>
</nav>

<?prg
  local cError := pvalue(1)
  local cHtml  := ''

  if valtype( cError ) == 'C'

    cHtml := '<div class="alert alert-danger">'
    cHtml += '<strong>Error !</strong> ' + cError
    cHtml += '</div>'

  endif

  return cHtml
?>

<form action="{{ Route( 'auth' ) }}" method="post">

<div class="card mb-3 " style="max-width: 540px;">
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
<div class="row no-gutters">
  <div class="col-md-4 container_logo">
    
  </div>
  <div class="col-md-8">
    <div class="card-body">
      <h5 class="card-title">Autentication</h5>
      <div class="card-text">

        <div class="form-group row align-items-center red">

          <div class="col-12">
            <label for="user">User</label>
            <input type="text" class="form-control" id="user" name="user"
placeholder="User login... (demo)">
          </div>

        </div>

        <div class="form-group row align-items-center red">

          <div class="col-12">
            <label for="psw">Password</label>
            <input type="text" class="form-control" id="psw" name="psw"
placeholder="Password (1234)">
          </div>

        </div>

        <div class="form-group row align-items-center text-center">

          <div class="col">
            <button type="submit" class="btn btn-primary"><i class="fas fa-
sign-in-alt"></i> Login</button>
          </div>

        </div>

      </div>
    </div>
  </div>
</div>
</form>
```



Mercury

Modelo/Vista/Controlador


Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Y he creado un pequeño fichero css → login.css con este contenido

```
.card {  
    margin: 0 auto;  
    margin-top: 50px;  
    float: none;  
    margin-bottom: 10px;  
    box-shadow: 5px 5px 5px grey;  
}  
  
.container_logo {  
    display: flex;  
    height: 100%;  
    margin: auto;  
}  
  
.logo {  
    margin-left: auto;  
    margin-right: auto;  
    width: 75%;  
}
```

Hasta aquí debemos ya conocer y encajar estas piezas. El resultado es el siguiente






Autentication

User

Password

 Login



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Bonito no ? Bueno en este caso parte del mérito es del Bootstrap → <https://getbootstrap.com/>, pero la conclusión al llegar aquí es que TODA la lógica de la aplicación sigue funcionando, no se ha modificado ni controller, ni modelo, ni router, validator, ... SOLO hemos cambiado la view que teníamos hiper simple con un poco de bootstrap, eso es todo y esta es parte de la magia del MVC cuando se entiende. Cada pieza hace su trabajo, independientemente de la otra. Bootstrap nos ha ayudado a dejarlo bonito, pero solo ha "pintado". Nosotros hemos construido todo el proceso que hay detrás y mueve el sistema, Debemos separar todos estos conceptos.

Si algún día quiero usar otro framework para maquetar como por ejemplo materialize → <https://materializecss.com/> , simplemente cojo esta view de login y la retoco. Todo el resto de la aplicación funcionará igual !!! 😊

WebServices

Llegando a esta parte del manual, nos queda hablar de como podemos hacer un webservice y será una tarea muy fácil con todo lo que hemos asimilado.

Nos basaremos en los webservices RESTful sencillamente por que son los mas fáciles de gestionar, mas potentes y porque los últimos años han emergido siendo los mas populares y usados.

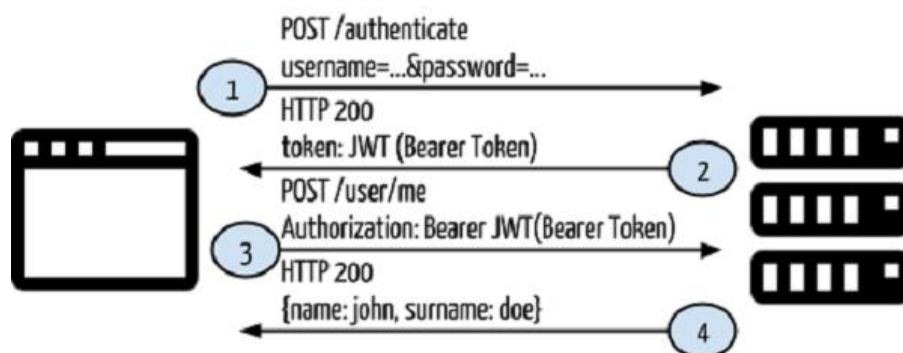
Básicamente un webservice es como indica la palabra, un servicio que te da una web, podríamos de momento centrarnos en consultas. Nosotros conectaremos a una ws por ejemplo para consultar la hora, el tiempo, ...hasta ws corporativos o que se necesitan una autenticación para consultar datos digamos mas sensibles.

Funcionamiento básico del WS

Básicamente el funcionamiento de un sistema de ws se basa en tokens. Nosotros vamos a reproducir el mismo sistema usando tokens y basando el standard de autorización OAuth 2.0 y es de tipo "Bearer", lo vemos más adelante.

Cuando nos conectamos al servidor que nos provee de ws el flujo es el siguiente:

- Nos autenticamos normalmente con user/psw
- El servidor si Ok nos devuelve un Token
- Cada petición que hagamos adjuntaremos en la cabecera: Authorization: Bearer JWT
- El servidor cheque q token es ok y procesa petición



Método GetByState() – Listado de clientes por estado

Nos volvemos a centrar al ejemplo que hemos realizado con "search" que buscábamos los clientes de un determinado "state". El proceso ya quedo resumido de la siguiente manera:

- Recogida de parámetros
- Validación de los parámetros
- Proceso de los datos con el modelo
- Respuesta pasando a una view los datos

El WS es prácticamente lo mismo ! solo cambia que antes enviábamos el error o el resultado a una view y ahora lo enviaremos directamente al navegador en un formato específico, en este caso en formato json.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

El formato mas usado por su flexibilidad, tamaño, etc,... es json por lo que mostraremos como realizarlo.

Crearemos un controller nuevo que llamaremos ws y crearemos el mismo método que teníamos en el anterior, GetByState(). El resultado seria este

```
METHOD GetByState( oController ) CLASS WS

local oCusto      := CustomerModel():New()
local oValidator  := TValidator():New()
local hRoles      := { 'state' => 'required|string|len:2' }
local hFormat     := { 'state' => 'upper' }
local hData       := {}
local hResponse   := {}
local aRows

//  Recupero Datos -----

hData[ 'state' ] = oController:oRequest:Get( 'state' )

//  Valido datos -----

if ! oValidator:Run( hRoles )

    hResponse[ 'success' ] := .F.
    hResponse[ 'error' ]   := oValidator:ErrorString()

    oController:oResponse:SendJson( hResponse )

    return nil

endif

oValidator:Formatter( hData, hFormat )

// -----

aRows := oCusto:RowsByState( hData[ 'state' ] )

hResponse[ 'success' ] := .T.
hResponse[ 'state' ]   := hData[ 'state' ]
hResponse[ 'rows' ]    := aRows

oController:oResponse:SendJson( hResponse )

RETU nil
```

Si comparamos el código de [GetByState@customer.prg](#) con [GetByState@ws.prg](#) solo cambia el tipo de respuesta que da el controlador. Por un lado envía a la view y por otro responde directamente enviando los datos en formato json. Para responder en formato json usaremos el método del objeto oReponse:SendJson() tal como se explico anteriormente.



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

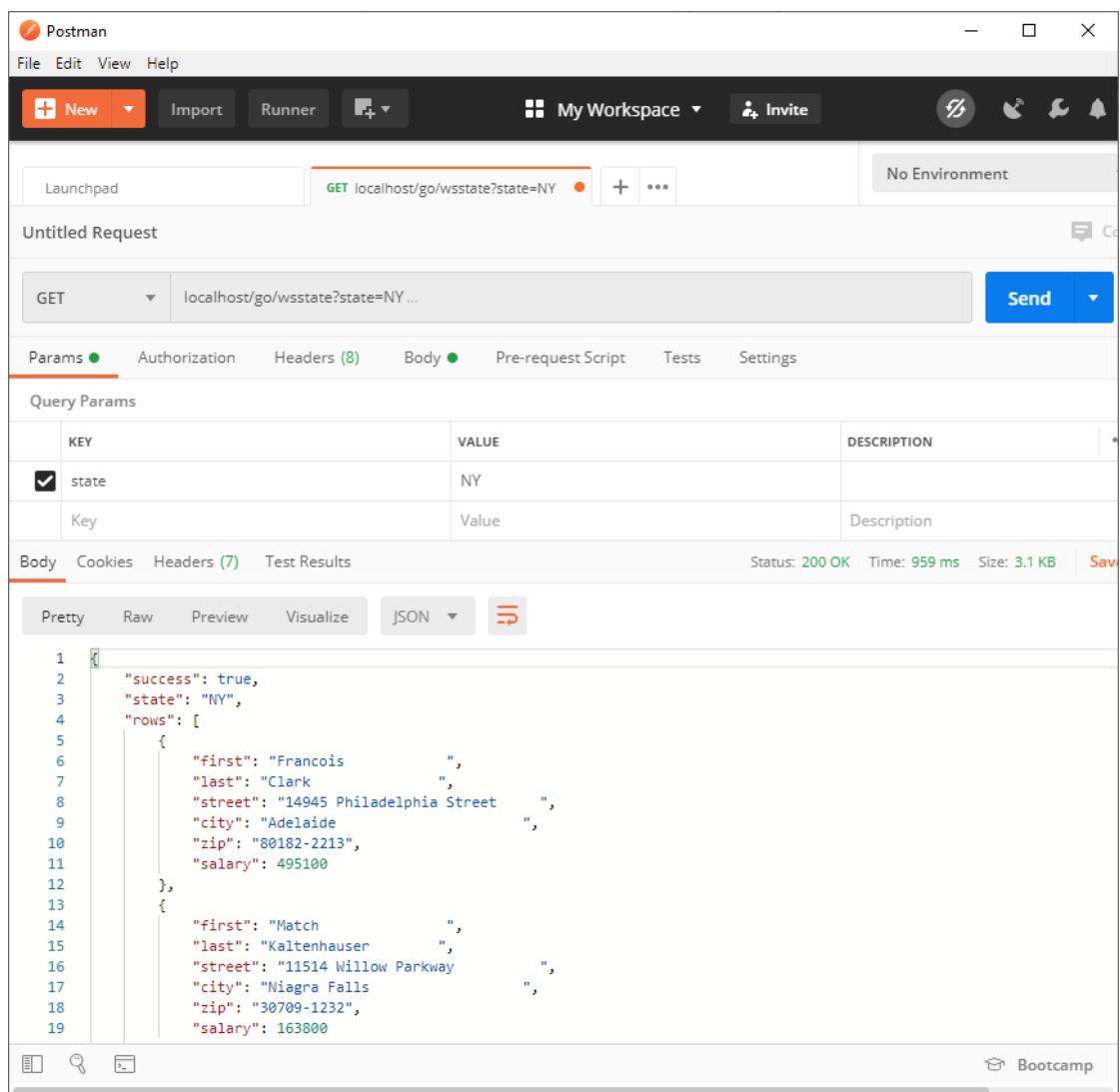
También elijo un tipo de respuesta que a mi me gusta, el json tendrá este formato:

- success (true/false)
- error (en el caso que success = .f.)
- rows (resultado del proceso)

Sabemos como funciona nuestro módulo, pues solo debemos crear su enrutamiento en el index.prg

```
DEFINE ROUTE 'wsstate' URL 'wsstate' CONTROLLER 'getbystate@ws.prg' METHOD 'GET' OF oApp
```

Vemos que la url que elegimos es 'wsstate'. Ya solo nos queda testearlo con alguno de los múltiples programas que existen. Elijo postman para realizar el test.



Podemos comprobar que el sistema nos responde y nos da la respuesta correcta en json. En el caso de poner una entrada errónea el sistema responde lo siguiente

Query Params

	KEY	VALUE
<input checked="" type="checkbox"/>	state	NYtres
	Key	Value

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ↗

```

1  {
2    "success": false,
3    "error": "Error authentication"
4  }
```

Tenemos ya la primera parte del ws preparada, ahora nos falta la falta de autenticación en el caso de que queramos proteger nuestro ws y solo lo dejemos usar a quien nosotros autoricemos.

Método Auth() - Creación de sistema autenticación

Nos basaremos en el método Auth que creamos en el controller Access.prg por que el concepto es el mismo. Una de las cosas que hemos de tener en cuenta es que devolveremos siempre las respuestas en formato json. Siguiendo este patrón, definiremos nuestro sistema con una estructura, en que definimos la key success para saber si bien o mal y las otras key de información. En este caso para la autenticación, devolveremos:

- Success (true/false)
- Error (en el caso que success = .f.)
- Token (en el caso que success = .T.)

```

METHOD Auth( oController ) CLASS WS

    local oValidator := TValidator():New()
    local hRoles      := { 'user' => 'required|string|maxlen:8|minlen:1' , 'psw' => 'required' }
    local hFormat      := { 'user' => 'lower' }
    local hData        := {=>}
    local hToken       := {=>}
    local hResponse    := {=>}
    local aRows
```



Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

```
// Recupero Datos -----  
  
hData[ 'user' ] := oController:oRequest:Post( 'user' )  
hData[ 'psw' ] := oController:oRequest:Post( 'psw' )  
  
// Valido datos -----  
  
if ! oValidator:Run( hRoles )  
  
    hResponse[ 'success' ] := .F.  
    hResponse[ 'error' ]   := 'Error authentication'  
  
    oController:oResponse:SendJson( hResponse )  
  
    return nil  
  
endif  
  
oValidator:Formatter( hData, hFormat )  
  
// Comprobar identidad -----  
  
if hData[ 'user' ] == 'demo' .and. hData[ 'psw' ] == '1234'  
  
    // Entro datos opcionales, que los puedo consultar cada vez que entren  
    // No se ha de poner informacion sensible...  
  
    hToken := { 'in' => time(), 'user' => hData[ 'user' ] }  
  
    // Entro datos opcionales, que los puedo consultar cada vez que entren  
    // No se ha de poner informacion sensible...  
  
    hToken := { 'in' => time(), 'user' => hData[ 'user' ] }  
  
    // 120 = 2 minuts; default 3600  
  
    cToken := oController:oMiddleware:SetAuthenticationToken( hToken, 120 )  
  
    // Preparamos respuesta  
  
    hResponse := { 'success' => .t., 'token' => cToken }  
  
else  
  
    hResponse := { 'success' => .f., 'error' => 'Error authentication.' }  
  
endif  
  
oController:oResponse:SendJson( hResponse )  
  
RETURN nil
```



Página 70 de 76



Método New() - Validación del token

Antes hemos de proteger nuestros ws para que funciones con sistema de autenticaciones. Seguiremos el mismo sistema que describimos antes, pero antes un matiz a tener en cuenta. Nuestro WS que hemos definido tiene 2 métodos:

WS	
Auth()	→ Sistema de autenticacion
GetByState()	→ Método de consulta

El sistema Auth() es un método publico que no necesita autenticaciones. Siempre podremos acceder para solicitar un token, mientras que GetByState() será un método privado que necesita previamente una autenticación. ¿Como debemos trabajar esta situación?

Como vimos en el tema de autenticación usábamos este preprocesado para autenticar

AUTENTICATE CONTROLLER oController ERROR VIEW 'login'

Hora el sistema es el mismo pero con una modificación:

- Debemos indicar que, en caso de error, en lugar de enrutarlo a una view, devuelva una respuesta json
- Indicarle que si el método que nos solicitan es el auth() (solicitud de token), no hace falta autenticar.

AUTENTICATE CONTROLLER oController VIA 'token' EXCEPTION 'auth' ERROR JSON
--

Esta es la única diferencia. En caso de error, el sistema envía un json con el siguiente contenido

```
{ 'success' => .f., 'error' => 'Error authentication' }
```

Podemos redefinir el mensaje de error añadiéndolo después de ERROR JSON

AUTENTICATE CONTROLLER oController VIA 'token' EXCEPTION 'auth' ; ERROR JSON { 'mierror' => 'No autorizado' }
--



Mercury

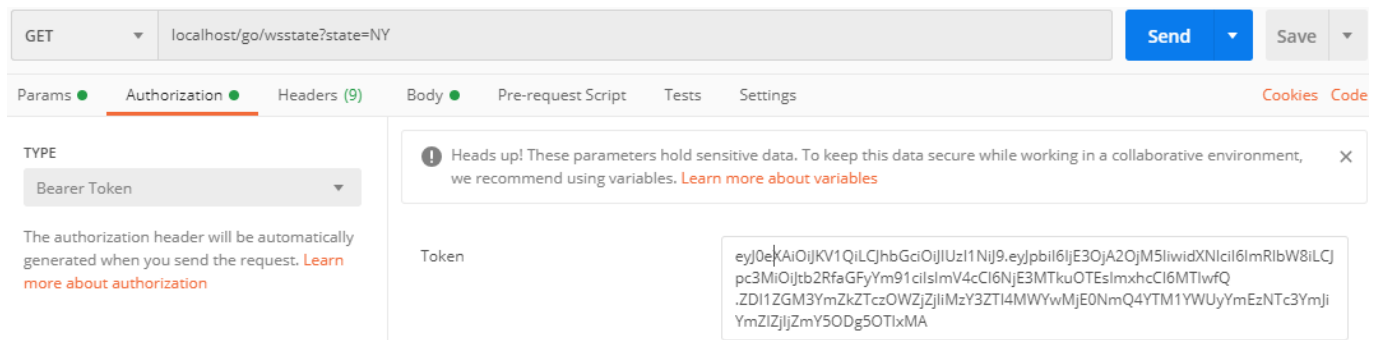
Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

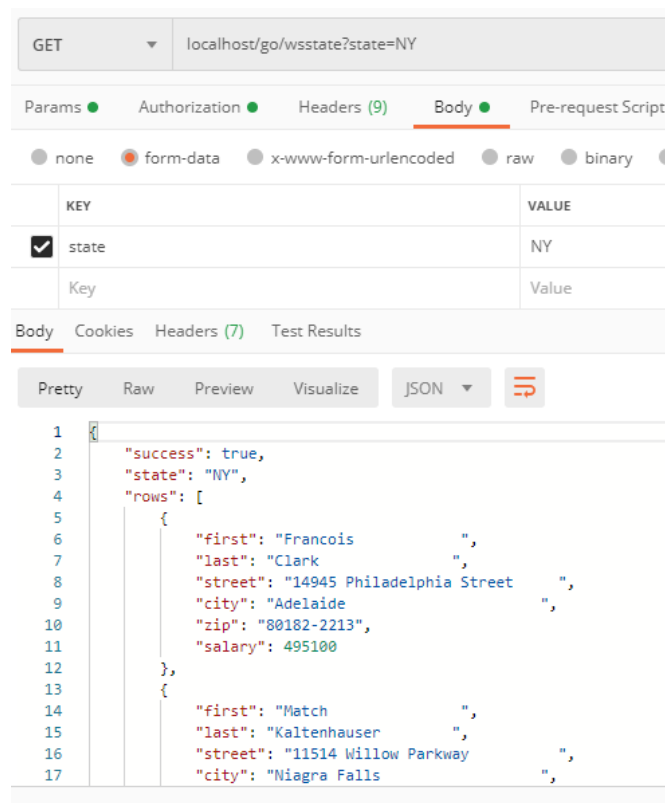
Pues ya solo queda probar el último paso y es la de crear la petición al ws. Deberemos pasar:

- Token que nos han dado antes para podernos autenticar
- Parámetro state que necesita el ws para saber que estado queremos
- Petición via GET

En el postman recordad de especificar token



Y los parámetros





Y ya tenemos nuestro primer WS con sistema de autenticación preparado y listo para colgarlo en nuestro servidor de internet 😊

Recordad que cuando hemos montado el ejemplo, al crear el token le hemos dado un validez de tiempo de 120 seg. Esto quiere decir que pasado este lapsus de tiempo el sistema nos debería dar error.

```
Pretty  Raw  Preview  Visualize  JSON  ↗
1  {
2    "success": false,
3    "error": "Error authentication"
4  }
```

Método Data() – Ver datos incrustados en el token

Para finalizar el capítulo dedicado a los WS podemos ver los datos del token en cualquier momento, recordad que habíamos incrustado datos adicionales a la hora de crearlo como hora de entrada y user, que son optativos y evidentemente podemos o no poner lo que queramos.

Hay una manera de recuperar esta información por si deseamos reutilizarla una vez que esta autenticada la petición y es usando el método

```
local hData := oController:oMiddleware:GetDataToken()
```

Pues tan fácil como crearnos un método ws al que llamaremos que nos muestre esta información. Crearemos el siguiente método:

```
METHOD Data( oController ) CLASS WS
    local hData := oController:oMiddleware:GetDataToken()
    oController:oResponse:SendJson( hData )
RETU nil
```

Recogemos los valores del token y los devolvemos via json, mas fácil imposible. Ya solo nos queda definir el enrutado

```
DEFINE ROUTE 'wsdata' URL 'wsdata' CONTROLLER 'data@ws.prg' METHOD 'GET' OF oApp
```

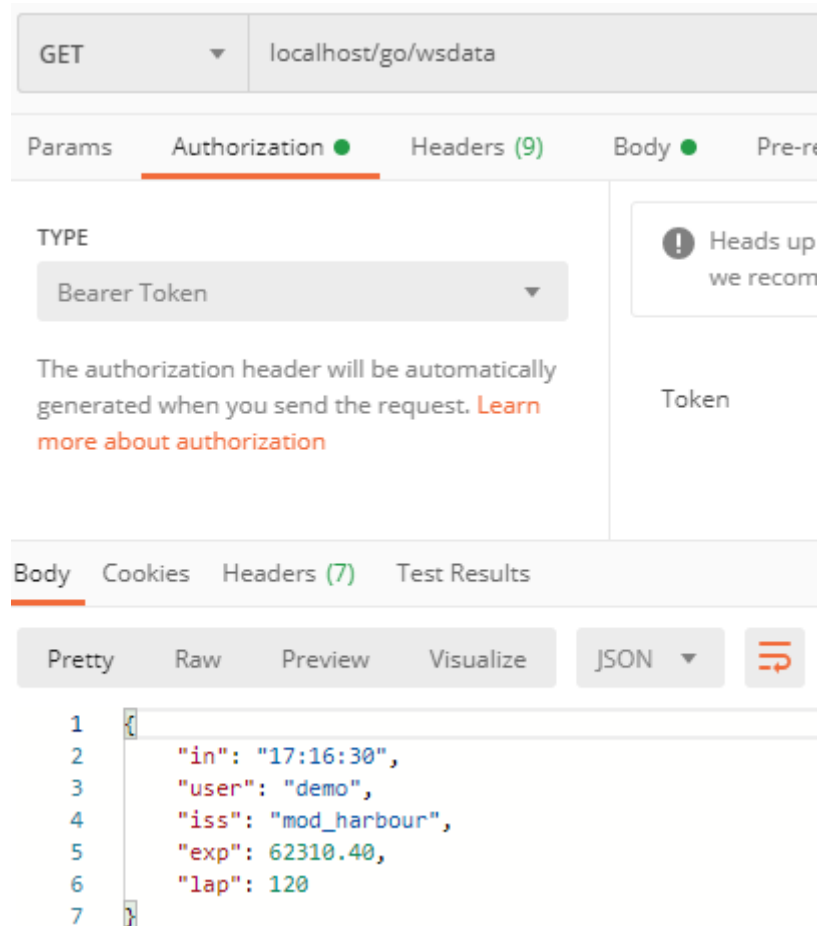


Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Y solo lo necesitamos probar con Postman, recordad que el token sea válido.



Podemos comprobar los 2 parámetros que definimos (in, user) mas otros 3 que son de uso interno del sistema. Ya tenemos todo lo necesario para crear tantos WS como queramos

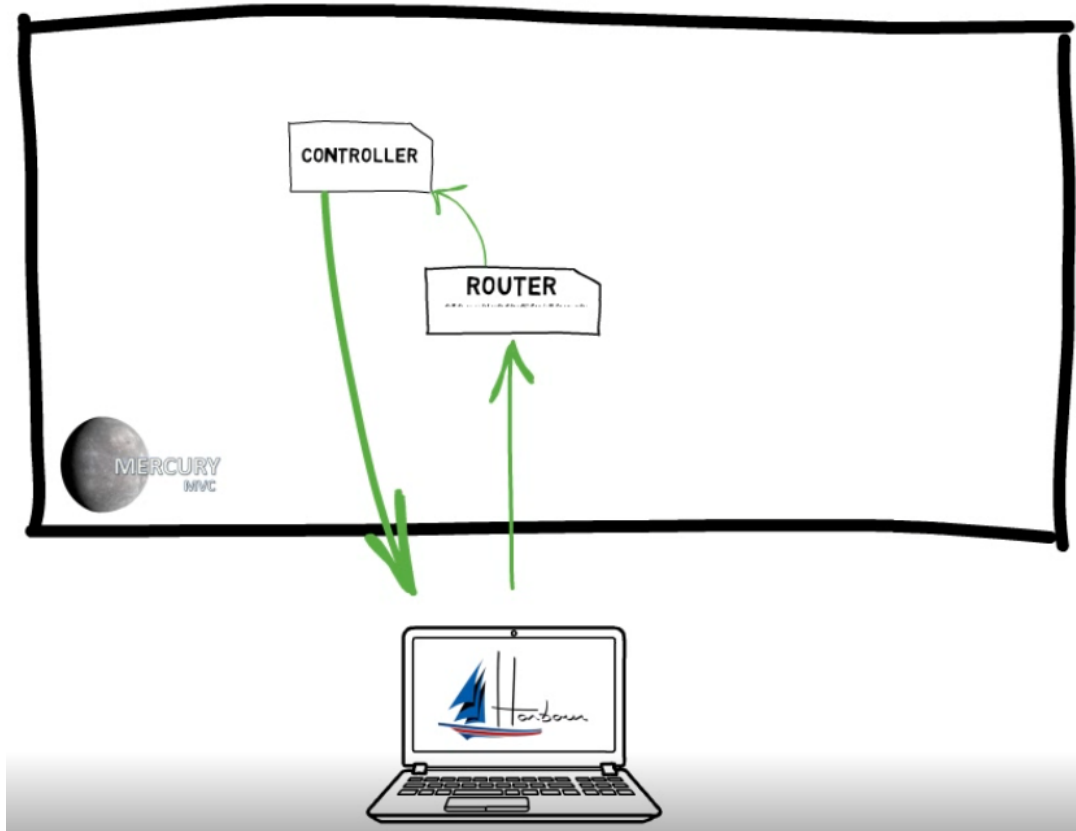


Mercury

Modelo/Vista/Controlador

Autor Carles Aubia
Fecha 28/07/2019
Versión 1.02

Conceptualmente tendríamos este escenario que ya lo explicamos en el apartado de controller y que ahora debemos entender completamente → <https://youtu.be/VnwM4kQKKmw>





Conclusiones

La Web es un monstruo que te come cuando la quieres dominar. Es necesario tener una disciplina a la hora de programar y seguir un buen patrón como el MVC si nos queremos dedicar a realizar aplicaciones serias y profesionales.

Al principio, como todo, es algo nuevo, quizás muy lioso, y a veces cuesta de entender, pero a medida que vamos aprendiendo y probando el sistema nos vamos a ir dando cuenta que es el camino. A medida que vayamos poniendo módulos a nuestro sistema, veremos como la estructura planteada aguanta perfectamente, hay buenos cimientos y sabemos donde ir a buscar cada cosa.

Espero que en este punto hayas podido encajar todas las piezas para que os hagáis una idea de como enfocar vuestras aplicaciones. Parece casi de locos para muchos que venimos de entornos de programación como Windows todo lo que debemos tocar para crear una web (esto es solo el principio 😊), pero a medida que vayamos entendiendo estas técnicas nos ayudara en nuestros proyectos y agradeceremos sus bondades.

Mercury es un framework construido en Harbour que trabaja como la mayoría de los frameworks para MVC que se usan actualmente con los lenguajes más populares

Harbour está en la Web !

Hemos llegado para quedarnos...

“Programar es fácil, hacer programas es difícil”

© Carles Aubia Floresví, 2019-2020