



# **Mercuy**

## **Model/View/Controller**

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1



**Harbour for Web** 

*Translated by Andres*





# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

Preamble .....	3
Mercury .....	4
Our system configuration .....	5
App(). Starting... ..	6
TRoute .....	7
TController .....	9
TRequest .....	12
TValidator .....	13
TResponse .....	14
TMiddleware .....	14
View .....	15
A.- HFW macro substitution .....	16
B.- PRG .....	19
B.1.- pickup by number parameter .....	20
B.2.- variables collection via one Setter/Getter .....	22



# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

## Preamble

I would like to discuss a few things with you before you start. We are a group of young people who took a season working in the world of madmen. Many of us from old systems, Clipper, after Harbour.

For professional reasons one has to jump to other given the technological evolution that we are privileged to live and I happened systems also make many changes.

Harbour for me is definitely the magical language of my life, it's special. He had long without programming but I've always been there beside me. After several meetings and meetings with many of you one of the major issues raised was that Harbour stayed behind, obsolete, sad. We lacked the big step, make the jump to the Web, like all popular languages that exist today.

Mr. Antonio Linares took it out once again and got the hat. Mod\_harbour the first seed was already set. It will work ?

Many languages have passed, but Harbour continues. We are the last to board the train from the web. Many go ahead, everyone. But we are scratchers code with "dozens" of years of experience and now is the time to get us all our knowledge and contribute it to the community. We have to make the jump and watch on equal terms to others. We just started but we talked with the beast.

Now we just need to put these first foundation and the basis of our beloved Harbour. That encourage people and see what is slowly becoming a reality. I think we are experiencing a new turning point and it will be exciting.

I have tried to build Mercury watching others do, great, those who succeed, trying to imitate the way they are now working million people worldwide. It is a base, it is clear and does not even reach 1% of the power of the great frameworks, but I want to see if our Harbour is patient and we can start running.

I do not know if maybe use a lot or work for them, but I try to contribute in some way in our community bringing my experience and try to encourage many others to make it bigger.

Guys ... (youth), Harbour is magic☺. I encourage you to try, participate and enjoy our new Harbour.

Charly Aubia - 2019



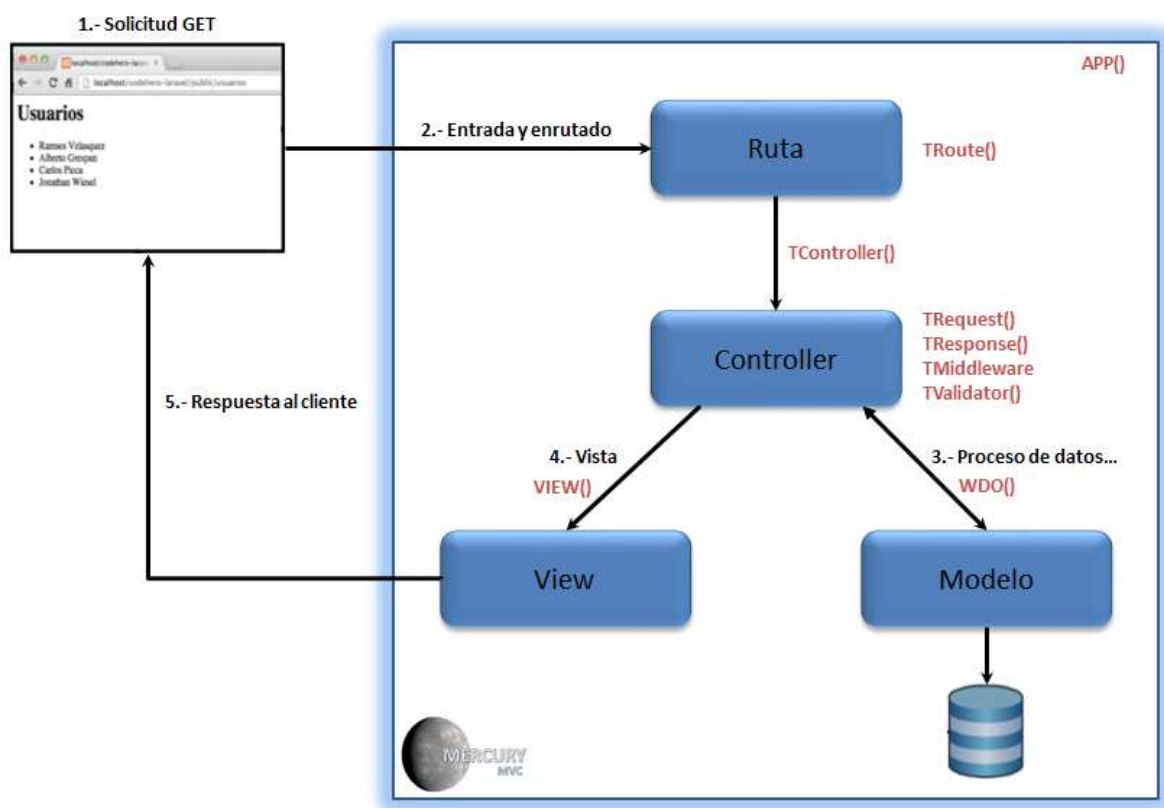
# Mercury Model/View/Controller

Author Carles Aubia  
Date 28/07/2019  
Version 0.1

## Mercury

**Mercury** It is an engine of MVC applications is part of mod\_harbour and allow us to efficiently structure the design of a Web application made with Harbour.

Following the standards of the great frameworks, we take their design concepts to apply to our source code and they will use classes provides us Mercury.



In order to use Mercury, when the program will have to specify the load module as follows:

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

This is all !

We assume that the libraries will be in the/lib



## Our system configuration

The MVC system always has a point of entry for the URL. That means that if our app is located in localhost/hweb/apps/myapp, every time we localhost/hweb/apps/myapp/miprogram.prg, localhost/hweb/apps/myapp/folder/test2.prg, localhost/hweb/apps/myapp/folder1/ folder2/ test3.prg ... our system will access by index.prg to be defined in our system with .htaccess

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

It is also important to define the following environment variables that will serve us time to manage paths in the application. Assuming we have our application in the folder/hweb/apps/ minimvc, would read:

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

These allow us to easily change the paths of the system.

We can define other parameters according to our convenience, but a file .htaccess base for our initial purpose might be as follows:

```
# -----
# CONFIGURATION ROUTES PROGRAM (Relative to DOCUMENT_ROOT)
# -----
SetEnv PATH_URL "/hweb/apps/mvc_mercury_hrb"
SetEnv PATH_APP "/hweb/apps/mvc_mercury_hrb"
SetEnv PATH_DATA "/hweb/apps/mvc_mercury_hrb/data/"

# -----
# Prevent read files in the directory
# -----
Options All -Indexes

# -----
# Default page
# -----
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>
```



## App(). Starting...

We begin designing the application statement App() function that will return an object of class TAPP(). This class is responsible for starting the system, paths, tracking, logs, errors .... internally create different classes that can then be used in the design, such as a router, request, response, ... the we will see as we need them.

Finally, before finalizing our program, we execute the method: Init() to start our whole system.

Thus we have the program begins and ends with the initialization of TAPP(). Might be:

```
// -----  
//      {% LoadHRB ( '/lib/core_lib.hrb')%}           //      core functions  
//      {% LoadHRB ( '/lib/tmvc_lib.hrb')%}           //      Mercury MVC system  
// -----  
  
FUNCTION Main()  
  
    LOCAL oApp    := App()  
  
    ...  
    ...  
    ...  
  
    //      The system started  
  
        oApp:Init()  
  
RETU NIL
```

We have reinforced the base of index.prg file but still does nothing.



## TRoute

As you all know the basis of MVC to hear the requests it receives the web and process this request routing (directing) the request to one place or another. This is where we will create our routes in our system and will be in the index.prg file.

To create the routes we will use the oRoute object found in oApp. The oRoute object has the following methods

### Map (<method> <id>, <map>, <controller>)

Map a route

<Method>

GET/POST/PUT/DELETE

(Currently only supports GET/POST)

<Id>

Identifier mapping. To use from anywhere in our application and we will find a route by id

<Map>

Mask our url. It will consist of a fixed part and a variable that can be default parameters. If our base URL is localhost/hweb/apps/minimvc, we can then add the different options that we allow to enter our application. Pe imagine that we want to create an entry for a list and we want the call parameter list. The map will "list". The url would be: localhost/hweb/apps/minimvc/list and this will cause a reaction in our system by running the driver you specify. For example if we consult further indicating a user id number, this id would be a variable parameter. If we call this action pe userinfo and map ID id serious "user/(id)". The system knows that we can make a pe entry

All variables parameters will brackets.

To indicate an action to our base map url will be "/"

<Controller>

Driver name to run when the request is routed. Our system will load the program you define. A controller will be a class with several methods. Imagine you have a controller that handles process 2 actions: list (list), see id (info) and we call this driver neighbors.

The base structure of this controller would pe

```
CLASS Neighbors
```

```
    METHOD New() BUILDER
```

```
    METHOD list()
```

```
    METHOD info() // Application Menu
```

```
ENDCLASS
```

```
METHOD New (o) CLASS Neighbors
```

```
SELF RETU
```



## Mercuy Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

```
METHOD list(o) CLASS Neighbors  
  
RETU NIL  
  
METHOD info(o) CLASS Neighbors  
  
RETU NIL
```

When we define in the Map the controller, this will be preceded by @ and the name of method we want to run. Suppose we create a route that if we specify neighbors/list we want to execute the method list of our neighbors controller. Should we say "[list@Neighbors.prg](#)"

Creating our first example in the system, create a route in case you do not enter anything in our url, execute a default controller and perform an initial screen. Our driver default welcome and our method will be called. Serious definition as follows:

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

If we run our application the system shows the following error, provided that oApp: IShowError this to .T. (Default value)



Our system can not find the driver default.prg. All drivers will be found by default in the/src/controller





## TController

We will create our default.prg handler with the following code

```
CLASS Default

    METHOD New() CONSTRUCTOR

    METHOD Welcome()

ENDCLASS

METHOD New( o ) CLASS Default

    RETU SELF

METHOD welcome( o ) CLASS Default

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

    RETU NIL
```

And back to refresh your browser



And we have our website and we have advanced the concept

In this example from the Method for the controller we print a message with ? processing an output of a string on the screen, but, as a rule, the outputs will be generated with the oResponse object or tell the oController:View( <cFile> ) to redirect to a view and you will be responsible for generating the output . But for practical purposes and quick check, we can use the?.

We'll see later the subject of View, but if we do not want to generate a kind view, oResponse output methods will have to provide us with other output formats.



# Mercury

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

Methods controller always receive oTController object as a parameter, with the following datas:

oController	Description
oRequest	TRequest object. It will serve to examine all variables concerning entry. see TRequest
oResponse	Tresponse object. It will serve to generate outputs. see Tresponse
oMiddleware	TMiddleware object. We will create our safety to the driver. see TMiddleware
CAction	Action sends the router
hParam	Hash parameter that sends the router
: ListController()	Controller information. For the purposes of debugueo, see what we get.
: ListRoute()	List of paths defined in index.prg. For the purposes of debugueo.
: View (<cFile>, ...)	Run the view <CView> by passing if any, the parameters for use from the view

We could create two methods within prg default to query the incoming information and routes with ListRoute() and ListController().

```
METHOD Info (oTController) CLASS Default
    oTController: ListController()

RETU NIL

METHOD Routes (oTController) Default CLASS
    oTController: ListRoute()

RETU NIL
```

And create routes in the index.prg

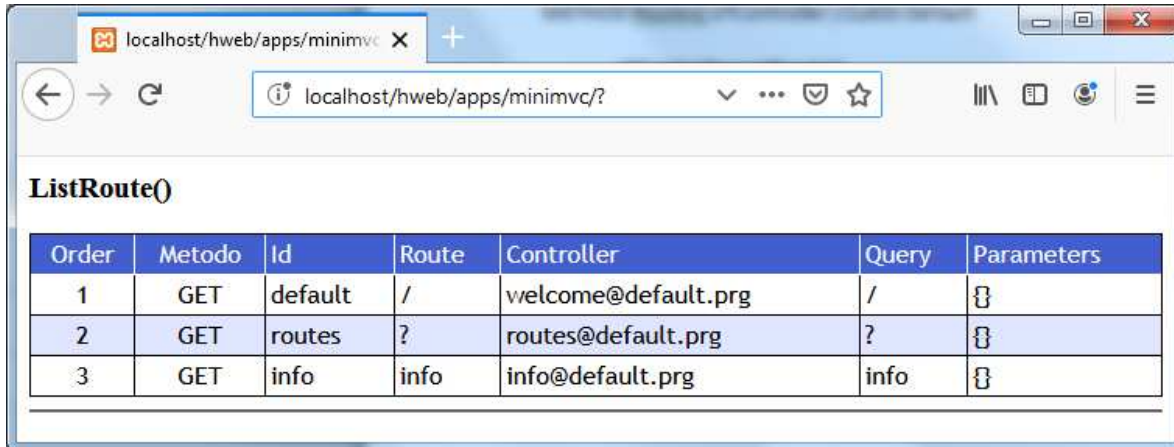
```
// Basic pages...
oApp:oRoute:Map( 'GET', 'default' , '/' , 'welcome@default.prg' )
oApp:oRoute:Map( 'GET', 'routes' , '?' , 'routes@default.prg' )
oApp:oRoute:Map( 'GET', 'info' , 'info', 'info@default.prg' )
```



# Mercury Model/View/Controller

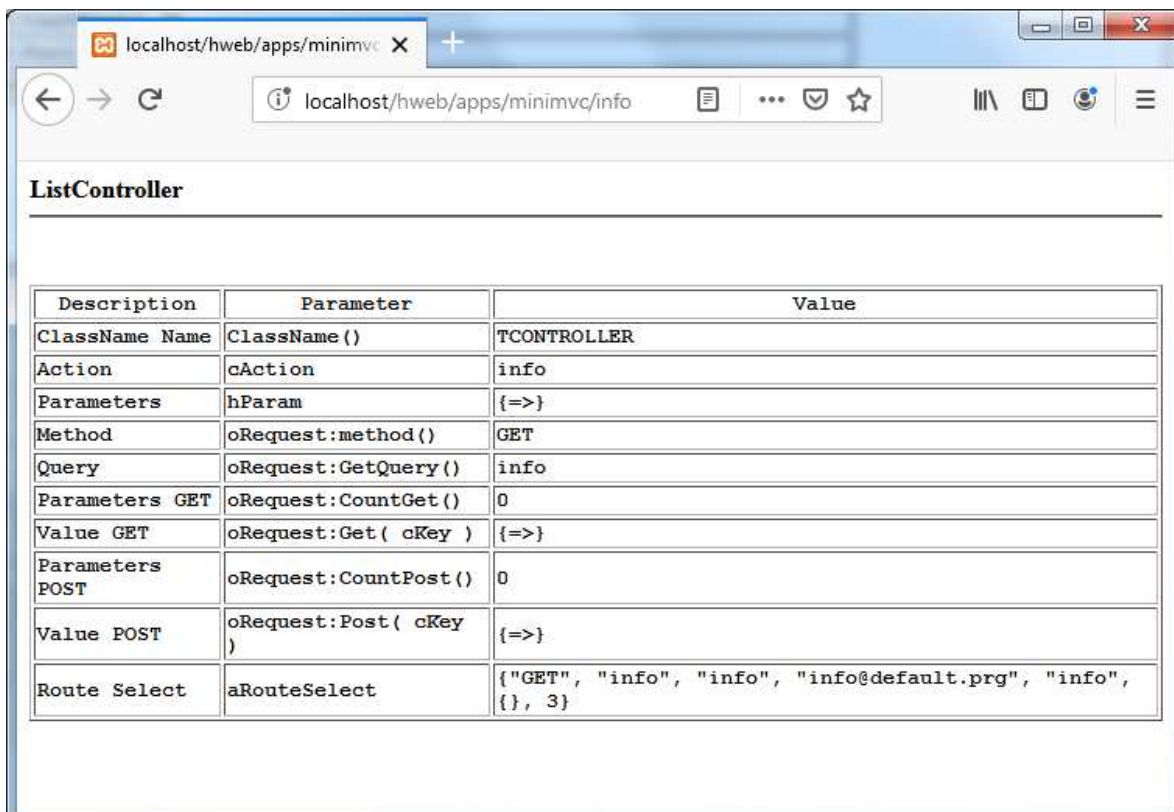
Author Carles Aubia  
Date 28/07/2019  
Version 0.1

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



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

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



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



# Mercury Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

The controller is part of one of the most delicate. Is the part of processing the request and roughly receive the input data that will manage it with the oRequest object, process data using many times the data models and move to the View (View) data to "paint" the screen

When the controller method is executed, we have seen that receive the oController. The first thing we do is look at the parameters we receive the request. For example if I ask a client data, it is necessary to receive at least one parameter that identifies this client, id imagine it.

As we designed our system, we know that method will be used: get, post or both ....

The object oController offers 3 direct methods to access this data according to the method:

oController: GetValue ( 'id', 0, 'N')

oController: Postvalue ( 'id', 0, 'N')

oController: RequestValue ( 'id', 0, 'N')

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

<cKey>            Parameter name

<cDefault>      Default value. Default is an empty string

<cFormat>        If we want to format the data already. They can be Harbour variable types: 'C', 'N', 'D'

## TRequest

With these functions and we can perfectly recover the data, but if you want more detail we can access them with direct methods of oController object: oRequest

oRequest - Methods	Description
Method()	Method which has made the request
Get (cKey, uDefault, cType)	Recover value via GET
GetAll()	
CountGet()	
Post (cKey, uDefault, cType)	Retrieve Value via POST
Postall()	
CountPost()	
Request (cKey, uDefault, cType)	Retrieve Value via REQUEST
RequestAll()	
GetQuery()	Query retrieve the URL
GetUrlFriendly()	
GetCookie (cKey)	Cookie recover



## TValidator

Once recovered the correct parameter would validate input data: If you have the expected format, length, allowed values, ... For this purpose we will use the Tvalidator class(). The Validator class that validate the parameters we have recovered based on rules. For example, imagine that in the case of id want it to be a required parameter and enter only numbers. The rules will go into a hash and can specify as many as you want and in this case would be something like this:

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

Create an object Tvalidator

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

And the concept is going to validate variables with rules. If all OK continue within our Controller, we will give you an answer but customer. This response may be a view or other response (we'll see later) as a json, xml, ... If an error occurs, the TValidator class will return by the method: ErrorMessage() an array of all the errors Have been found. The usual technique is eg if the validation fails is sent to a view with errors and the view, if we wish, and show these errors.

So we would validation system as follows:

```
//      Data validation

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

```

Here we hooked the subject of replies oResponse we mentioned previously. Imagine that we are creating an api and response rather than an HTML page, we return in json. In this case we would use the method oResponse: SendJson() to generate this output

```
//      Data validation

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

```



## TResponse

oResponse - Methods	Description
SetHeader (cHeader, uValue)	Create a header output
SendJson (uResult, nCode)	Create headers and data output JSON. nCode default = 200
SendXml (uResult, nCode)	Create headers and data output JSON. nCode default = 200
SendHtml (uResult, nCode)	Create headers and data output JSON. nCode default = 200
Redirect (cUrl)	Redirecting to URL
SetCookie (cName, cValue, nSecs)	Generate a cookie

## TMiddleware

Middleware will be the guardians and protectors of our controllers. The we may use as appropriate and the aim is to verify the authorization of the request to execute a method

Basically we will middleware in the new() method of the controller. Here we can check the Middleware() JWT type to see if a user is authenticated. It is the moment we have activated because we do not have sessions running.

The concept is: If I run the middleware() on the controller and not what happens we can redirect to a view, eg the default screen, login ... If the middleware authorizes the controller to execute the request method.

Serious construction something like

```
METHOD New( o ) CLASS MyApp

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

    IF o:cAction $ 'default'           //    Modulo que NO se validan: publicos, defaults,...

        // 'No middleware...'

    ELSE

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

    ENDIF

RETU SELF
```

The middleware it can recover the object oController: oMiddleware, pe

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



## Mercuy Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

Middleware type 'jwt' is that we will use when a user is authenticated. We will have a controller to validate access. In the event that authenticates properly execute the method:

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

HTokenData where data are bearing the token validation and always be able to recover every request and nTime time validation of this token between each request. Each time a request is made the token is regenerated again with the set time.

If a user leaves the system and generates a logout execute the method:

```
oMiddleware: CloseJWT()
```

## View

Part of the views is the last process in which we finally get the results and "paint" with html. We have said before that it is possible that since the party controller make a reply to the client with oResponse in a data format as pe Json and this does not bode data, but returns an answer ready, but this point of views is aimed at as we set out in html

The oRequest object has a method: View (<cFileView>, ...) which will be responsible for carrying our HTML file, process it and return it to the client browser. Files <cFileView> default are located in the/src/view so that there will place a extension.view, why call from the controller might look like oController: View ( 'home.view' CNAME) .

The file will contain pure and simple html and there is no secret that use standard html, js, css ...

```
<html>

    <h2>Hello Vista !</h2>

</html>
```

The peculiarity of these \* .view is that we can mix our code to optimize their use and create our process of creation and design of a page using 2 programming techniques:



## A.- HFW macro substitution.

In which we indicate an enclosed function between `{{...}}` → `{{Time()}}` in which the system processes all have the file before returning. A simple example would be:

```
<html>

    <h2>Hello Vista !</h2>

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

</html>
```

We can use all the functions Harbour at our disposal to compose our web !!!

We also use a special feature that is View (`<file.view>`). This gives us even more flexibility to our pages. Imagine you have an HTML header on every page that does the same: to declare html page, upload css, js files, pe ...

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

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

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

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

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

</head>
```

Imagine we have 3 standard pages: `page1.view`, `page2.view`, `page3.view`. These three pages must have also defined the above code cabecera.El put it into a file `head.view`. and now we could build 3 pages like this:





# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

### Test1.view

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

And this in the 3 files. Ready !!!

Now imagine that we have the typical bar bootstrap a nav in 3 pages with this code

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

We could put this mini code (template) in a file → nav\_default.view

Now, in the 3 pages we have created add as follows:

### Test1.view

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

And ready: 3 pages with this type of code. When head.view modify or nav\_default.view automatically affect 3 pages. This is the concept of the template that well combined will give us more productivity and efficiency in our codes.



## Mercuy Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

Just missing complement this part with proper use of our resources to load path. If we have this code:

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

  ...

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

  ...

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

  ...
</head>
```

Do you remember our variable start `oApp = App()`? We can refer at any point in our program with `App()` and with it access many of our application data. We could use it to define our code like this:

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

  ...

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

  ...

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

  ...
</head>
```

`App(): Url()` always give us the basis of our aplicacón path to use it referecnia in our definitions.



### B.- PRG

This is the other option of working in the views. It is inserted Harbour html code among all separated by tags `<?prg and?>`.

We can create us our data processing routines but in the end we have to return an HTML code to be inserted and susituirá the preprocessed code, eg.

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

    <?prg

        LOCAL cHtml := ""
        LOCAL nI

        cHtml += "<ul>"

        FOR nI := 1 TO 4

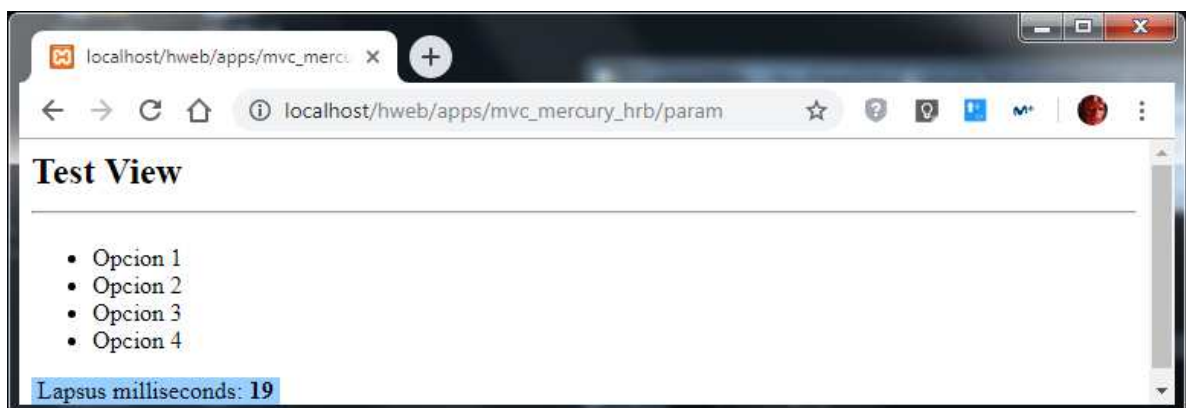
            cHtml += "<li>Opcion " + ltrim(str(nI))

        NEXT

        cHtml += "</ul>"

        RETU cHtml

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





# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

Collecting parameters from the controller

As we have explained, the controller is responsible for processing data and then passes them to the view to use them. We have 2 ways to do it

**B.1.- pickup by number parameter.** We know when we designed the controller that when we call the view method we can pass parameters, eg.

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge

    //    Tratamiento de datos

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

    //    Solicitud de Vista

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

RETU NIL
```

cName is the parameter 1 2 Nage, ...

At the hearing we can refer to them in this way

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

  <h3>Método parametros</h3>

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

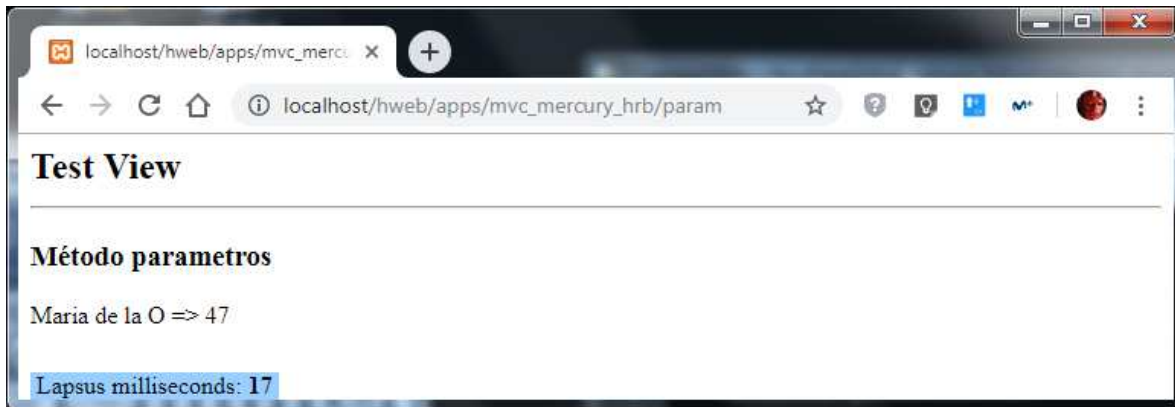
</body>
</html>
```



# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1





# Mercury

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

**B.2.- variables collection via one Setter/Getter**, Eg. the controller specify the data you want to use in the view with a setter → `_Set( <cVar>, <uValue> )`

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge, aData

    //    Tratamiento de datos

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

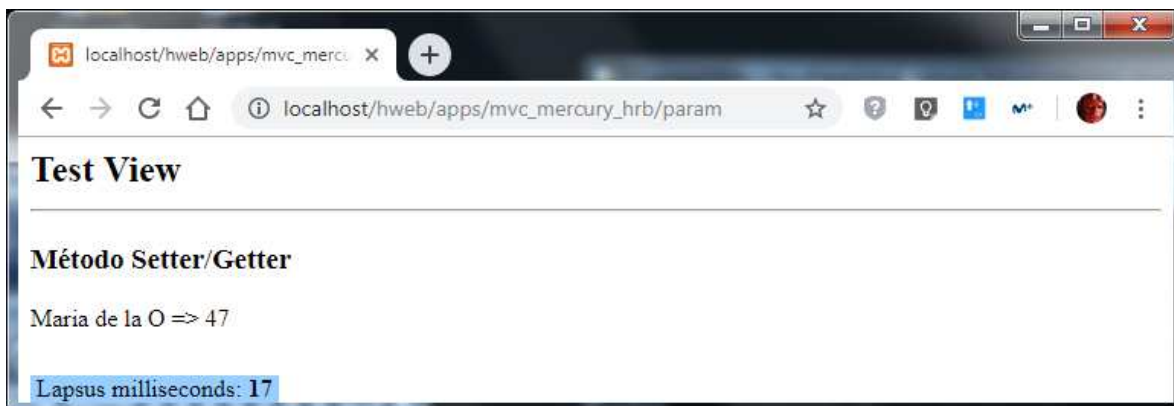
    //    Solicitud de Vista

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

    o:View( 'test_param2.view' )

RETU NIL
```

And in the view using `_GET (<cvar>)` function to collect this variable





# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1

We will use all together to assess differences

### Code controller

```
METHOD Test( o ) CLASS Param

    LOCAL cName, nAge, aData

    //    Tratamiento de datos

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

    //    Solicitud de Vista

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

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

RETU NIL
```

### Code view

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

    <h3>Método parametros</h3>

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

    <br><br>

    <h3>Método Setter/Getter</h3>

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

    <?prg

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

        FOR nI := 1 TO len( aData )

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

        NEXT

        RETU cHtml

    ?>

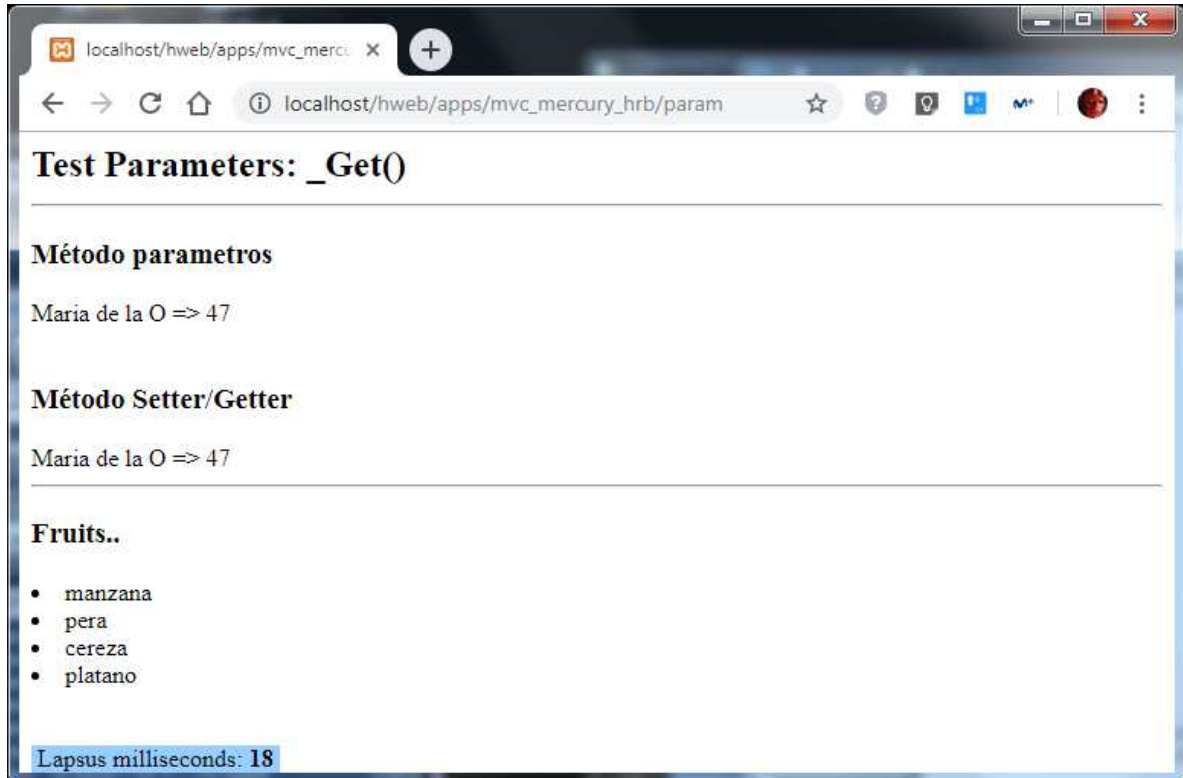
</body>
</html>
```



# Mercuy

## Model/View/Controller

**Author** Carles Aubia  
**Date** 28/07/2019  
**Version** 0.1



And that's all !!! The correct use of different templates with the use of macro substitution HFW and prg (embedded) code will give us a tremendous performance, reliability and maintainability our app.

MODEL - WDO

Not Yet !!!