Mercury
Model/View/Controller

Autor Carles Aubia

Date 10/06/2020
Version 1.02

Harbour for Web

mod Harbour

Autor	Carles Aubia
Date	10/06/2020
Version	1.02

Autor	Carles Aubia
Date	10/06/2020
Version	1.02

Preamble

I would like to discuss a couple of things with all of you before we start. We are a group of young people who have spent a season working in that crazy world. Many of us come from old systems like xBase, from Clipper, then Harbour.

For professional reasons one has to jump to other systems given the technological evolution that we have the privilege of living and I had to make many changes as well.

Harbour for me is without a doubt the magic language of my life, it is special. It took time without programming it, but I have always had it there, by my side. After several meetings and meetings with many of you, one of the big issues that was addressed was that Harbour was left behind, obsolete, sad ... We were missing the big step, making the leap to the Web, like all the popular languages that exist today.

Mr. Antonio Linares once again took it out of his sleeve and got it. The first seed of mod_harbour was already in place. It will work ?

Many languages have passed, but Harbour continues. We are the last to board the train on the web. Many go ahead, everyone. But we are code scrapers with "dozens" of years of experience and now is the time to take all our knowledge and contribute it to the community. We have to make the leap and look from you to you to the others. We have just started but we are already talking to the beast.

Now we just need to lay these first foundations and bases for our beloved Harbour. That the people be encouraged and see that little by little it is becoming reality. I think we are experiencing a new turning point and it will be exciting.

I have tried to build Mercury by seeing how others do it, the great ones, the ones that are successful, trying to emulate the way that millions of people around the world are working today. When I started it, it was another seed in this project, but after a year it has matured enough to give us confidence in this system and a way to start our projects effectively.

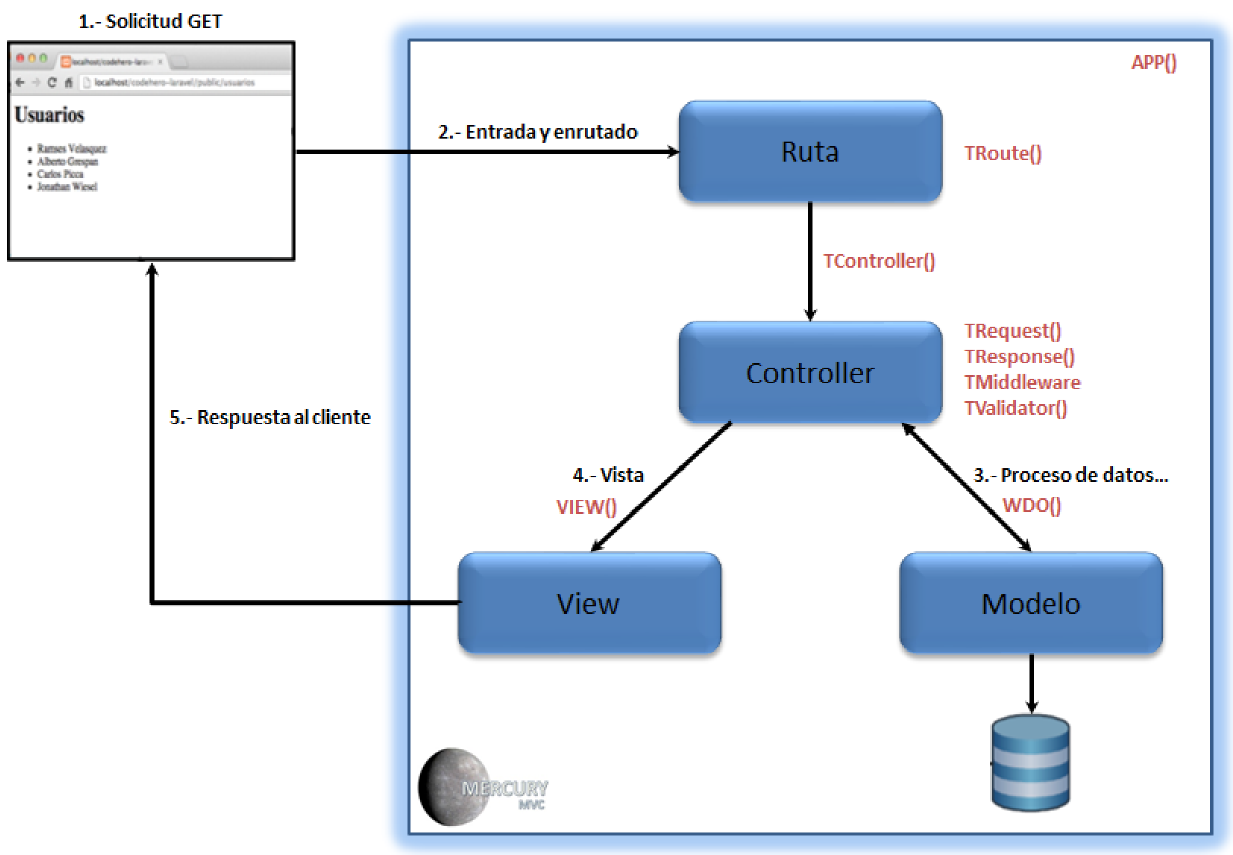
During this year I have also been able to verify how difficult it is for most to enter this programming environment and especially adapt to this new software architecture, but because we come from another programming culture and we know that we have to recycle ourselves. I know changes always cost.

I have reissued this manual to make it easier to follow and all the examples are ready to try. We start from 0, step by step, perhaps very easy for those who know the system, perhaps very difficult as we go, but we end with an authentication system, access control to the modules and the entire application closed. Colleagues ... Harbour is magical !. I encourage you to try, participate and enjoy Mercury to create your web applications and take the final step to go to the web.

Autor Carles Aubia
Date 10/06/2020
Version 1.02

Mercury It is an MVC application engine that is part of the mod_harbour and will allow us to efficiently structure the design of a web application made with Harbour.

Following the standards of the big frameworks, we try to follow the same line and concepts when designing the program.



We will explain step by step all these concepts, why, the advantages, the fit of one piece with another ...

How to use this manual.

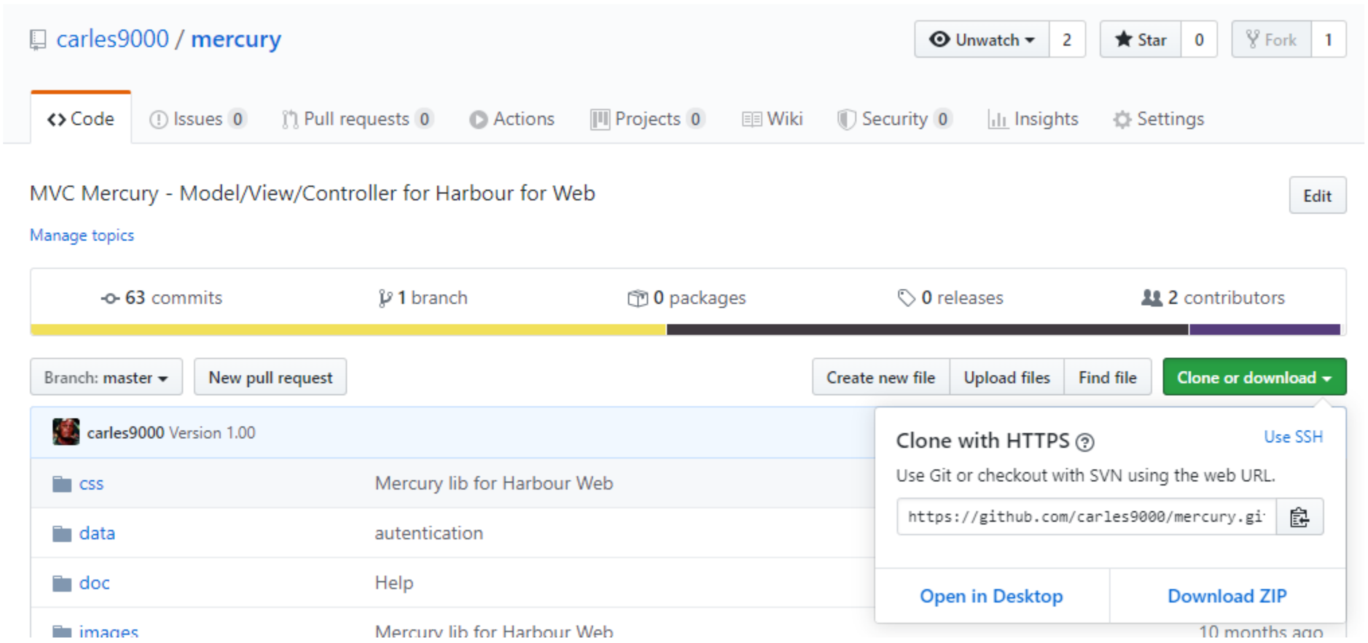
The best way to learn is by scratching and testing the examples. They have all been tested and the best thing to do is go step by step and copy / paste and try and understand why we are doing it. If we do it until the end of the manual, I assure you that you will understand and can start making your applications.

Autor Carles Aubia
Date 10/06/2020
Version 1.02

Installation

We start from the base, we already have mod Harbour installed and working correctly, but you can check on this page how to do → https://modharbour.app/compass/search/instalacion

The first thing to do is download the framework from https://github.com/carles9000/mercury.git doing a zip download


carles9000 / mercury

MVC Mercury - Model/View/Controller for Harbour for Web

63 commits | 1 branch | 0 packages | 0 releases | 2 contributors

Clone with HTTPS
Use Git or checkout with SVN using the web URL.
https://github.com/carles9000/mercury.gi

carles9000 Version 1.00	
css	Mercury lib for Harbour Web
data	autentication
doc	Help
images	Mercury lib for Harbour Web

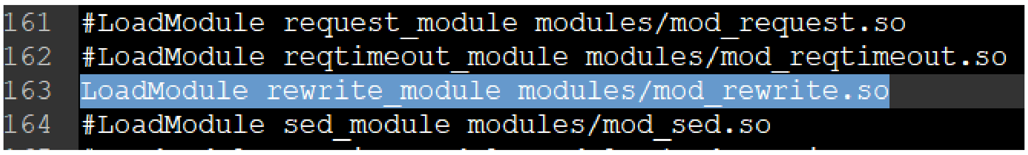
Or whoever uses git to make a clone → git clone https://github.com/carles9000/mercury.git

Autor	Carles Aubia
Date	10/06/2020
Version	1.02

Configuration of our system

1.- Activating Apache mod_rewrite

We have to parameterize Apache so that we can use the rewrite module. For this we will go to the configuration file httpd.conf and see that we have the line of the active module (uncommented)



```
#LoadModule request_module modules/mod_request.so
#LoadModule reqtimeout_module modules/mod_reqtimeout.so
LoadModule rewrite_module modules/mod_rewrite.so
#LoadModule sed_module modules/mod_sed.so
```

And add these lines

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

2.- Setting up our .htaccess file

Remember to shutdown and restart the server for these lines to take effect. What is achieved with these changes? We can redirect requests made to our server to an entry point. The MVC architecture always has the same entry point. This means that if our app is located in localhost/miapp every time we do localhost/hweb/miapp/miprog.prg, localhost/miapp/folder/test2.prg, localhost/miapp/folder1/folder2/test3.prg, ... our system will redirect access to localhost/miapp/index.prg.

To finish achieving this, in our project we will define the .htaccess file which is used to configure the environment of our application. In it we will add these lines

```
<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 at the moment for managing paths in the application. Starting from the basis that we have our application in the / hweb / apps / minimvc folder, it would look like this:

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

This will allow us easily if one day we change location adjust the system paths.

We can define the other parameters at our convenience, but a base .htaccess file for our initial purpose could be as follows:

```
# ----------------------------------------------------------------------------
# CONFIGURATION PATH APPLICATION  (Relative to DOCUMENT_ROOT)
# ----------------------------------------------------------------------------
SetEnv PATH_URL            "/hweb/apps/minimvc"
SetEnv PATH_APP            "/hweb/apps/minimvc"
SetEnv PATH_DATA           "/hweb/apps/minimvc/data/"


# ----------------------------------------------------------------------------
# Prevent them from reading the 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>
```

Summary: Once we have configured our apache with point 1, this part will no longer be touched. For each new Project we will adjust the paths of our .htaccess file as shown in point 2.

Creating our first PRG

File structure

We start from the base that we already have modHarbour installed and it is running perfectly. The method that we will use is valid in both Windows and Linux environments, but we will base ourselves in the Windows environment as there are more users than you use it. If you only have Apache installed or Xampp, we will create our directory for the project inside the htdocs directory, e.g. go → /htdocs/go and we will create the following structure

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

And finally at the root create our file .htaccess → /htdocs/go/.htaccess who would have this setting

```
# --------------------------------------------------------------------------
# CONFIGURATION PATH APPLICATION  (Relative to DOCUMENT_ROOT)
# --------------------------------------------------------------------------
SetEnv PATH_URL            "/go"
SetEnv PATH_APP            "/go"
SetEnv PATH_DATA           "/go/data/"


# --------------------------------------------------------------------------
# Prevent them from reading the 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>
```

This is our basic structure of our program. As we move forward we will add other directories if we need them, such as the data, include,...

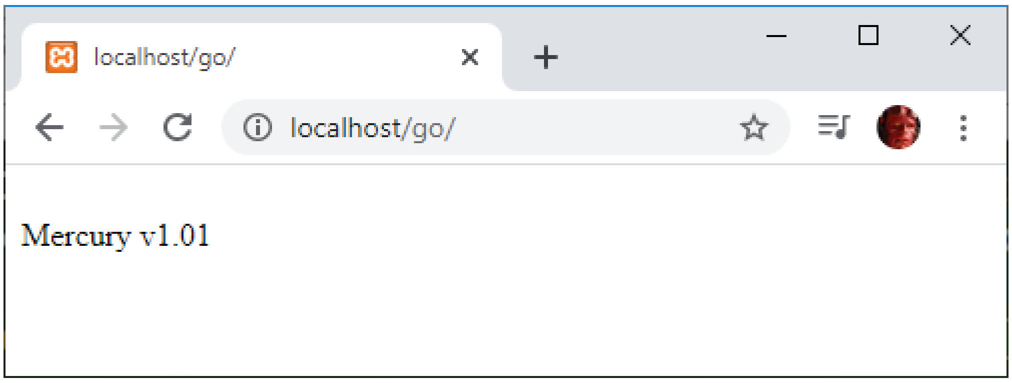
Now we can only test if everything is in order. The best way to check them is to create a program that loads the *mercury* and writes its version to us on the screen. If we get to this point it will mean that everything is in order

Main project file → index.prg

Our index.prg file will be the entry point to our application. The first thing we should do is check that we have *mercury* well installed. We will create the index.prg and add the following lines:

```
//     {% LoadHRB( 'lib/mercury/mercury.hrb' ) %}     //     Loading system MVC Mercury

function Main()

      ? MercuryVersion()

retu nil
```

If we go to our browser and type localhost / go, a screen similar to this should appear



```
Mercury v1.01
```

It is very important to see the first line that is where we load our mercury library, and that the syntax is

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

It is necessary to put the // of comments, because apache will execute before starting our program everything that is between {% and%}. It is a way of loading hrb modules and it works perfectly.

This is all !

Why MVC?

I think the Wikipedia entry is pretty well explained and defined:

Model-view-controller (MVC) is a pattern of software architecture, which separates the data and mainly what is the business logic of an application from its representation and the module in charge of managing events and communications. For this MVC proposes the construction of three different components that are the model, the view and the controller, that is, on the one hand it defines components for the representation of the information, and on the other hand for the interaction of the user. This software architecture pattern is based on the ideas of code reuse and the separation of concepts, characteristics that seek to facilitate the task of application development and subsequent maintenance.

The advantage of using this system is that once assimilated we will obtain great benefits in our programs and our maintenance will be better. We will know where we will have to go at all times and we will have everything well ordered and structured.

There is a learning curve that can cost us a lot to assimilate to those that we come from other programming environments, but I think it is the best system to successfully exit a web project. In the end we can affirm that this model follows the "divide and conquer".

At the beginning we will see that we cut everything into many files and we do not understand how we can carry out all our needs in a single file, we have to create perhaps 3,4,5 ... But we will see that everything fits when we reach the end of the manual.

Part 1 - Creating our first project

As I explained before, there are a lot of options when designing our application and the most important thing is to understand how the different pieces will fit together. As we have mentioned, each request we make to our app will go through an index.prg that will be the entry page to our app.

In the Index we will define our application, our map, our Router, which will be in charge of dispatching the different requests to our application. Anything that is not defined in this map cannot be executed and in summary all the different accesses will be programmed from here.

Router: Index.prg

We will start with the structure of the file index.prg, where we will start loading mercury, defining the application and creating our routes.

```
//      ------------------------------------------------------------------------------
//      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 aplication...'

                //      Config Routes

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

        //      System init...

                INIT APP oApp

return nil
```

This is the basis of our master file index.prg. We will describe the most important parts

- Loading our library mercury.hrb

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

- Loading the header file to be able to use Mercury commands

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

- Definition of our Application with the command DEFINE APP

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

- Creation of routes. It is the part where we define the different accesses to our modules from our url. The command is as follows:
-

```
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	It is an identifier that we give to our route. This identifier can be used from other points of the application to refer to its definition. For example, if we have a route defined like this: DEFINE ROUTE 'pedido' URL 'order' CONTROLLER do@orders.prg METHOD 'GET' OF oApp At any point of the program we can execute the Route function ('order') and this will return the defined url, in this case “order”
cUrl	It is the url defined for this route. Following the previous definition, if in the url of our browser we put “order” → localhost/go/ order, our application will know that we have to execute in this case the controller do@orders.prg, because we have it defined that it can be executed using the GET method (see cMethod). If we define the url as '/' it is telling our application that if we do not enter anything in our browser url, execute what we have defined → localhost/go DEFINE ROUTE 'hello' URL '/' VIEW 'hello.view' METHOD 'GET' OF oApp
cController	It will be our controller, our prg that will be in charge of executing the pertinent code for this action. These prg will be small classes, with their methods that we will invoke from the Router. In this case we tell the application to run our controller orders.prg and invoke the do() method
cView	It will be our view associated with this route. Sometimes we don't need any controller to process data, just display a view. DEFINE ROUTE 'hello' URL 'hola' VIEW “hello.view” METHOD 'GET' OF oApp

	If we put in our url 'hello' → localhost/go/hello, our system will know that it will have to execute our view 'hello.view'
cMethod	It is the GET or POST method that is used in html to define an access type and that we can use. There is a lot of documentation on the subjec thttps://es.stackoverflow.com/questions/34904/cuando-debo-usar-los-m%C3%A9todos-post-y-get Basically when we want to load pages we will use the GET method, while POST is used more for sending and updating data. Attention if we want to execute an Url, because we will have to define in the route that it is of type 'GET'. In the case that we want to define a route that is executed in any of the cases we can add ... METHOD "GET, POST"
oApp	APP object that we have previously defined

- At the end of the index.prg we will start the app with the INIT APP <oApp> command

Summary: We are creating a base where we define all the different accesses to our application, a map. This implies that as our application grows, we will add different routes in our index.prg

View

Our first view: hello.view

The view will be our part of the code that defines the page to be displayed in our browser, our html. Whether we like it or not, the web has its languages that we must use in one way or another, so at least basic knowledge is necessary to understand how it works. Basically we will use html, css and javascript, so it is important to explore them 😊

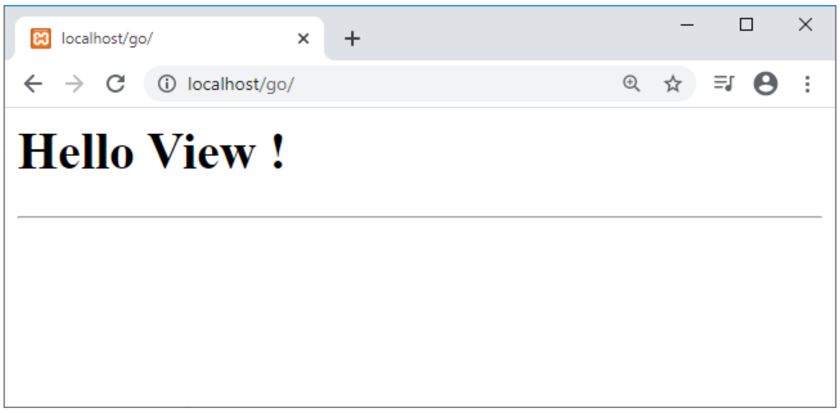
The views will be in the src / view directory and we can add folders to that directory in case we need them. One of the objectives of the MVC is to follow a hierarchy and order everything, so that later our maintenance is easier.

Following our example, we will create the file src / view / hello.view and put the following code

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

We already have a view created. Pure and hard html. They will be pages in which the coding will be in html, not in prg, although later we will see how to insert prg code in our views.

Well, we already have everything necessary to run our first MVC application in modHarbour. If we write in the url of our browser → localhost/go it should appear



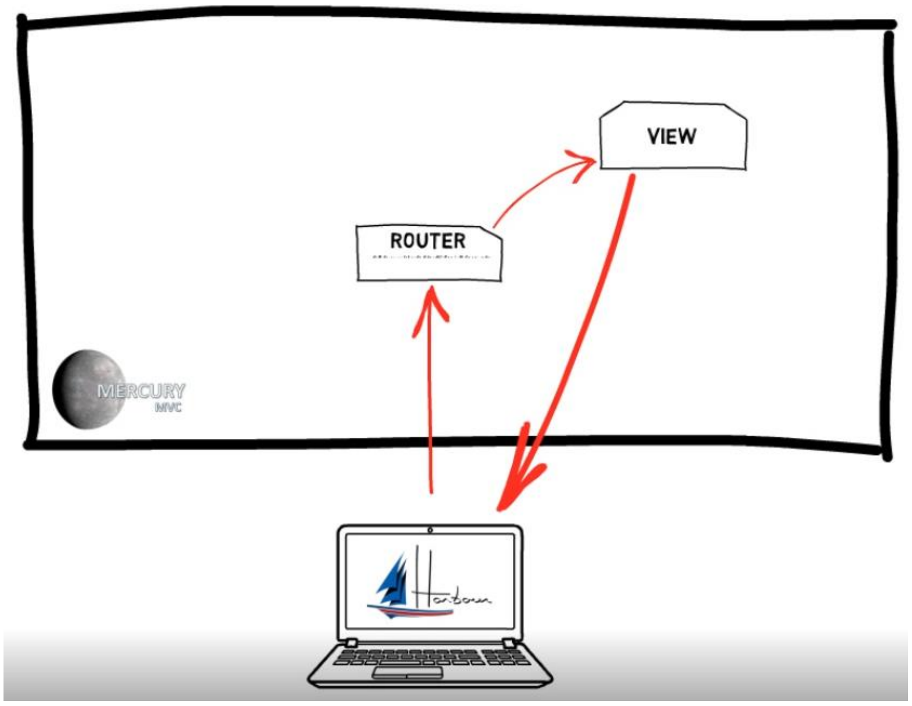
Hello View !

We already have our base to scale our application and keep all control.

Summary:

- Create a route that points to our view
    - `DEFINE ROUTE 'root' URL '/' VIEW 'hello.view'  METHOD 'GET' OF oApp`
- Definition of our view → hello.view

Conceptually we would have this scenario → https://youtu.be/N6fN_4pIdj4



The first question we can ask ourselves is why so much history without this code we can also do it

```
function main()

  ? "<h1>Hello View !</h1>"
  ? "<hr>"

return nil
```

You must think that we are designing the foundations for our application, in which strict order and control will be necessary and that, as we scale, our functionalities will not be a problem. MVC is a software architecture that is well tested and accepted in the developer community of the main existing platforms. If we understand this concept we can continue with our purpose.

To finish this first example we are going to create a simple page that shows us a link to another page, to see how to manage our router. The second page will have another link that will send us back to the first page.

One of the advantages of programming html, css, js, … is that we have millions of help pages, meaning that we will not be alone in this battle. For example, to find how to create a link, we can find an example from here → https://www.w3schools.com/tags/tag_a.asp

So we will create 2 pages that we will call view1.view and view2.view and we will define 2 routes in our index.prg following the procedure described.

```
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
```

Observe how we tell our application what we can use and what not, that is, at these times we can put the url go/, go/view1, go/view2. Any other entry will be ignored

Now we will define view1 as follows in order to see how we will “reroute” to view2

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

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

We look at the first concept of macrosubstitution in our html code using the {{...}} keys. The system will replace the Harbour code that we have between the keys. In this case we have our Route (<cRoute>) function which is the one that will return the associated url.

A Route () function tells our system what is the url that we have defined under the route "view2" and as we have defined in our router that is "view2.view". The system will do what it has to do but at the end it returns go/view2. That means that in the end the pre-compiled code would end like this:

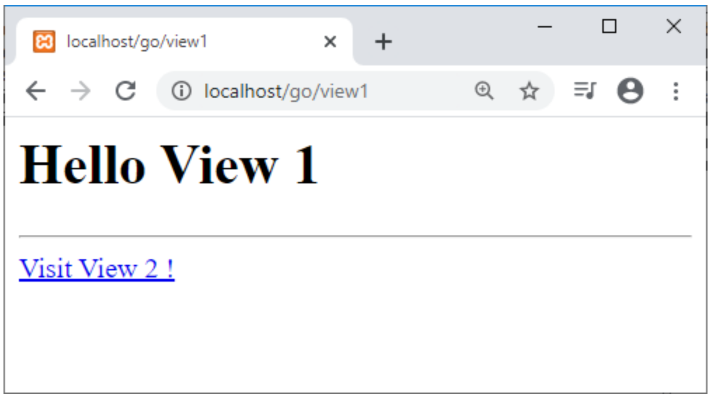
```
<h1>Hello View 1</h1>
<hr>

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

The system will always be guided by what we define in our Router. We do not have to worry about where the code is or url addresses, or anything. The Route will be in charge of mounting it. As our application grows, if we have e.g. 5 views that refer to a url, doing it this way, if one day we change the address in the route, it will automatically change in the 5 views, we should not worry about manually changing that new url in each view, this is one of the many virtues of working in this way.

Continuing with the example, we will create view2 as follows.

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

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

It is the same example, but on this page we refer to view1



Hello View 1

Visit View 2 !

And this is the concept in this section of the Router. If you try it you will see that you jump from one side to other and it is our router that will be in charge of controlling these urls.

This point is important because we must understand how, from the different points of our program, when we make a request to the server, the way to find these urls must be from the definition of our application map that we have created in index.prg , our "router".

Injecting PRG code

Within a view which we are encoding in html, we can inject prg code. The way to do it is by opening the <prg tag and closing ?> Inside we can put as much Harbour code as we want, the only condition is that it returns a string that will be inserted right where the tag starts.

```
<h1>Test PRG</h1>
<hr>
	<?prg
		local cHtml  := 'Now is ' + time() + ' of ' + dtoc( date() )

		return  cHtml
	?>
<hr>
<h3>Code html from prg section...</h3>
```

We can put as many prg code blocks as we want.

Injecting HTML code into PRG

And finally, to finish curling the curl, inside a prg code block, we can insert an html code block, so we don't have to mount html statements by concatenating strings. To inject code html into a prg block we will use

BLOCKS VIEW <v> [PARAMS …]

We have to pass a variable to the block which is where the code will be assigned. If we wish we could also pass variables from inside the prg to the block to be able to use them using PARAMS <mi_var>, <mi_otra_var>,…

In the case of passing variables to the BLOCKS VIEW, from inside we will use them by putting them inside the labels <$ … $>

Imagine this example injecting prg code and inside the prg code we want to mount an html output

```
<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>
```

```
<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>
```

In large chunks of code it is cleaner and easier to code by putting it all inside a BLOCKS VIEW

And with this we close the topic of views and as you can see we can go from mounting a pure and hard html code, to helping us by inserting code prg, to injecting html code, inside prg code. A range of options that we can use.

If you create the entries in the router of these two examples you will be able to execute them easily, and you will be able to verify that the system works correctly.

Internet browser → Router → View → Internet browser

Perhaps with what we already know you can already think that you have enough and perhaps depending on how far it can be. You can think that I create a view that we open a table, let's look for example the data of a record and show them in the same view. Let's do a test

We will create a directory in our project called data and we will put there our beloved table test.dbf that will be indexed by several fields including one by "state"

If you remember at the beginning in our .htaccess we define this environment variable

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

Now is the time to use it. When we use Harbour that will run on our server, we refer to our paths with all the real path as we always do, not with the relative one. This means that to know where we have our data in our project we could do:

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

This will surely point us if we have xampp installed at c:\xampp\htdocs\go\data

I am used to creating functions in the index that facilitate this path, because you are going to use them a lot. For example, in the index.prg I would add this function

```
function AppPathData()

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

This function is already visible from now on from controllers, views, models, ...

We are going to create a view that creates a web page and shows me the users of a state.

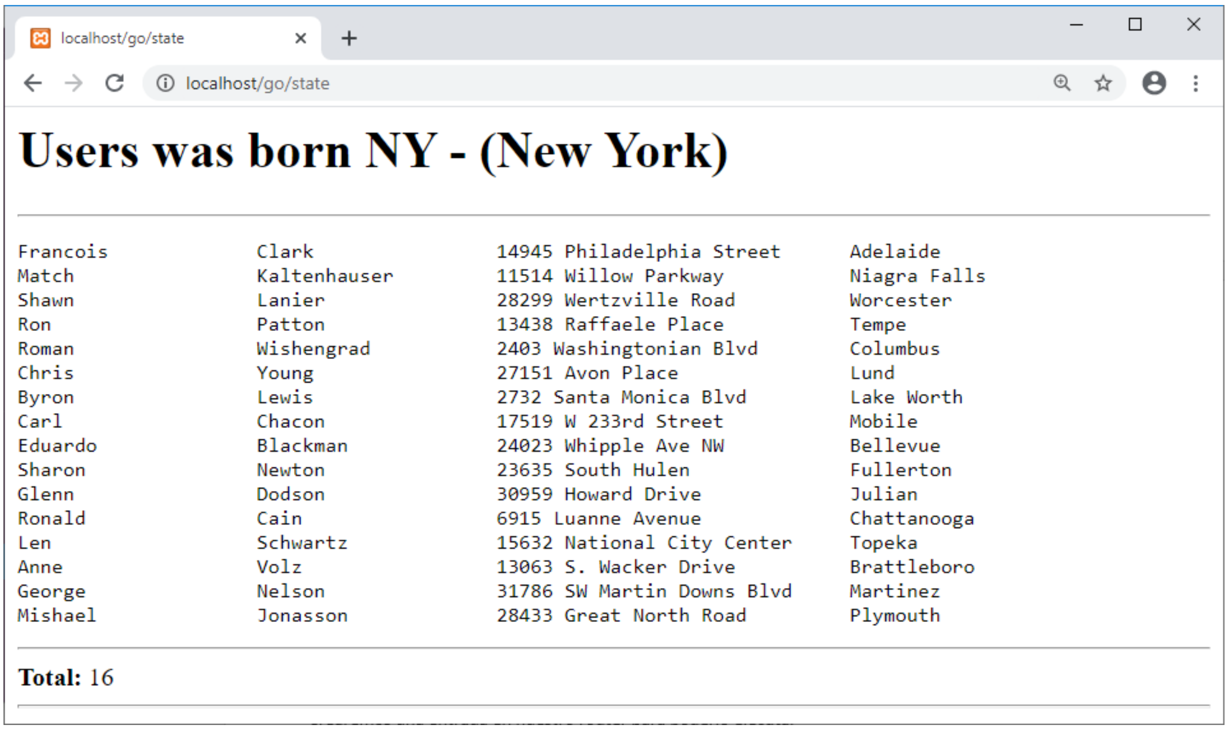
```
<h1>Users was born NY - (New York)</h1>
<hr>

      <?prg
            local nCount := 0
            local cHtml  := ''

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

            cAlias := Alias()

            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>
```

We will create an entry in our router to be able to execute it

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

And we just run → localhost/go/state



Users was born NY - (New York)

```
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
Mishael             Jonasson           28433 Great North Road        Plymouth
```

Total: 16

What do you think? Simple with a simple view to put all this functionality, but our MVC architecture does not want to follow this pattern, but rather another. With how easy we had it ☹

So where do we fail? We fail in the concept. A view has to be in charge of painting, drawing, assembling the page, ... but not assembling the logic of the program, processing data, opening tables, performing calculations, ... all of this is handled by our Controller!

Controller

The Controller is the one in charge of processing all our program logic, knowing what to do with what the user asks of us, creating queries, updates, calculations,.... This is the purpose of the controller and once it has finished its task it has several options of which we will see a few, but we will see 2 that are the most important:

a) Send the data to a view so that the web is set up, also using all this data 😊
b) Return to the browser in this case the data for example in json format

There are other options that are not going to be dealt with at the moment, such as sending the result in xml format, in a text file, ... The most important thing is to understand the part of the Controller.

The controllers will be in the src / controller directory and we can add folders to that directory in case we need them. As we discussed in the views section, one of the objectives of the MVC is to follow a hierarchy and order everything, so that later our maintenance is easier.

A controller is a class with several methods, it has a constructor and it is very easy to build allowing us to scale the program little by little in a very simple way. Let's imagine that we create a program to manage a virtual store. We could create a controller to manage everything related to products, another for orders, stock, ...

a.- Example of a controller that sends data to a view

Following the example of the view with the users of NY (New York) we are going to create a user controller that will have a method to query the users of a "state". The structure could look something like this

```
CLASS Customer

      METHOD New( oController )               CONSTRUCTOR

      METHOD GetByState( oController )

ENDCLASS

//------------------------------------------------------------------------------//

METHOD New( oController ) CLASS Customer

RETURN Self


//------------------------------------------------------------------------------//

METHOD GetByState( oController ) CLASS Customer


RETURN nil

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

As we can see we already have a module (Customer) that will be in charge of managing everything that refers to our customers....

If you observe the code you will see that all methods will receive an *oController* parameter with a series of properties that will help us throughout the process and that we will see during the process.

We have commented that once the controller processes data, it can scramble a response to the browser or create a request to generate an output view. When we need to send to a view we will use the method oController:View(<Name_View>)

Autor Carles Aubia
Date 10/06/2020
Version 1.02

The first thing we are going to do is create a method inside the controller that will do nothing and will call a View to mount a data entry screen and we will call this method e.g. Search ()

At this point we will see how we create an entry in our routes file index.prg.

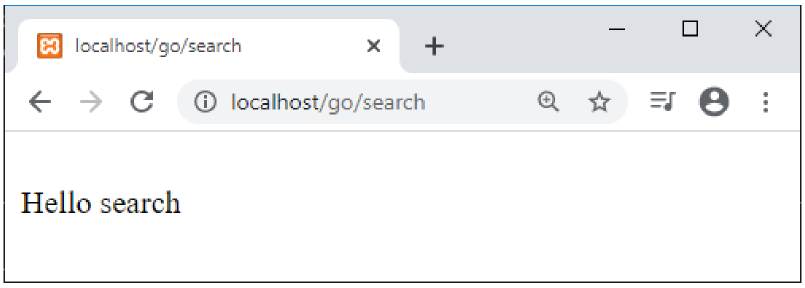
```
DEFINE ROUTE 'search'  URL 'search'  CONTROLLER  'search@customer.prg ' METHOD 'GET' OF oApp
```

The novelty here is that we use the CONTROLLER command and we indicate the name of the controller preceded by an @ and the method it has to execute → `'search@customer.prg`

Now we can add the Search () method in the controller to see if the request arrives when we enter the url → localhost/go/search

To test if it works we can momentarily put a ? 'Hello Search'

```
METHOD Search( oController ) CLASS Customer

       ? 'Hello search'

RETU nil
```



Hello search

Now it is about passing the 'Hello search' to a view, as we have said with the method oController:View()

```
METHOD Search( oController ) CLASS Customer


  oController:View( 'search.view' )

RETU nil
```

And we will create the view in our folder src/view/search.view

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

And with this small example we already see a process triangle:

Internet browser → Controller → View → Internet browser

Send parmeters from Controller → View

Before continuing with our example, we will explain the passing of parameters from a controller to a view.

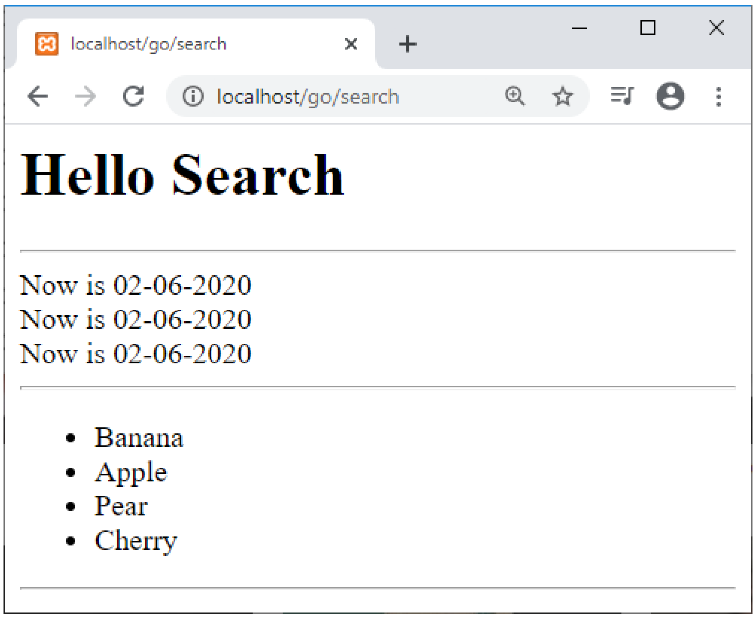
Let's imagine that we have 2 variables (a string and an array) that we have processed in our controller to finally pass it to a view. The syntax is oController:View(<Name_View>, <par1>, <par2>, ...)

```
METHOD Search( oController ) CLASS Customer

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


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

RETU nil
```

And in the view we are going to assemble the page collecting the parameters that the Controller sends

```
<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

              cHtml += '</ul>'

              retu cHtml
       ?>

<hr>
```

If we run the url localhost/go/search it will look like the following screen



```
Hello Search

Now is 02-06-2020
Now is 02-06-2020
Now is 02-06-2020

• Banana
• Apple
• Pear
• Cherry
```

The view code is to analyze and see how we can collect the parameters. If we observe the date we can collect it 3 ways

- Macrosubstitution putting between {{ ... }}
- Code block prg between <?prg ... ?>
- Macrosubstitution using the command PARAM <nParametro>

The second block that shows the fruits we see that it is a prg block, and we collect the 2 parameter (fruit array) and then we process it as we have seen in the views section.

There is a fourth way that we comment on it as information that works correctly but perhaps breaks the traditional method. It is from the Controller to prepare the variables that we are going to use in the view with: App:Set (<cNameVar>, <uValue>) as many times as we need and from the view to collect the parameter with App:Get (<cNameVar)

At this point we must already understand how a controller can process your data and send it e.g. to a view

b.- Example of a controller that sent the data to the browser

But we can also create a system that what we want is that the server returns the result only in data, for example in json format. This can be when we create a request for a page in Ajax and do not want another new page but add data. In this way we could create this type of request and add data to our screen.

To do it is very easy. The controller instead of sending the data to a view, it will do it to the browser. To do this, it will use the oController object: oResponse which is in charge of making the outputs with its different methods.

oResponse

oResponse - Métodos	**Descripción**
SetHeader(cHeader, uValue)	Create an output header
SendJson(uResult, nCode)	Create JSON headers and data output. nCode default = 200
SendXml(uResult, nCode)	Create JSON headers and data output. nCode default = 200
SendHtml(uResult, nCode)	Create JSON headers and data output. nCode default = 200
Redirect(cUrl)	Redireccionar a Url
SetCookie(cName, cValue, nSecs)	Create a cookie

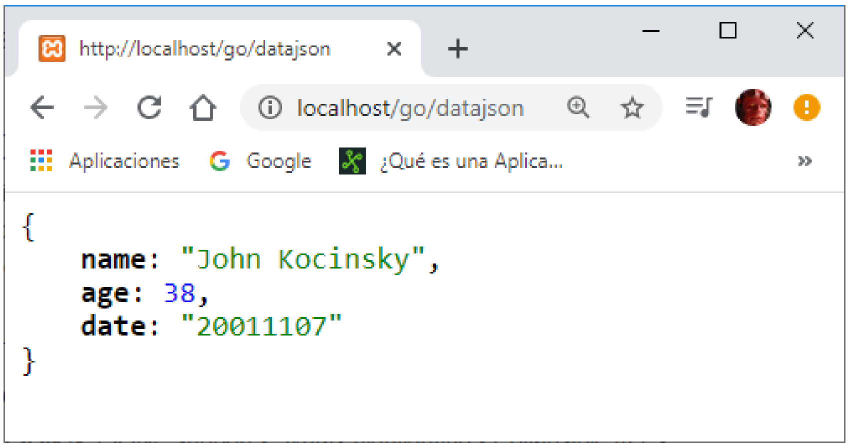
We create an entry in the Router:

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

And we create method data_json in controller dof this way.

```
METHOD data_json( oController ) CLASS Customer

  local hUser  := { 'name' => 'John Kocinsky', 'age' => 38, 'date' => CTod( '07/11/2001' ) }


  oController:oResponse:SendJson( hUser )

RETU nil
```

If we execute → localhost/go/ datajson, the data sent by our controller in json format should appear.

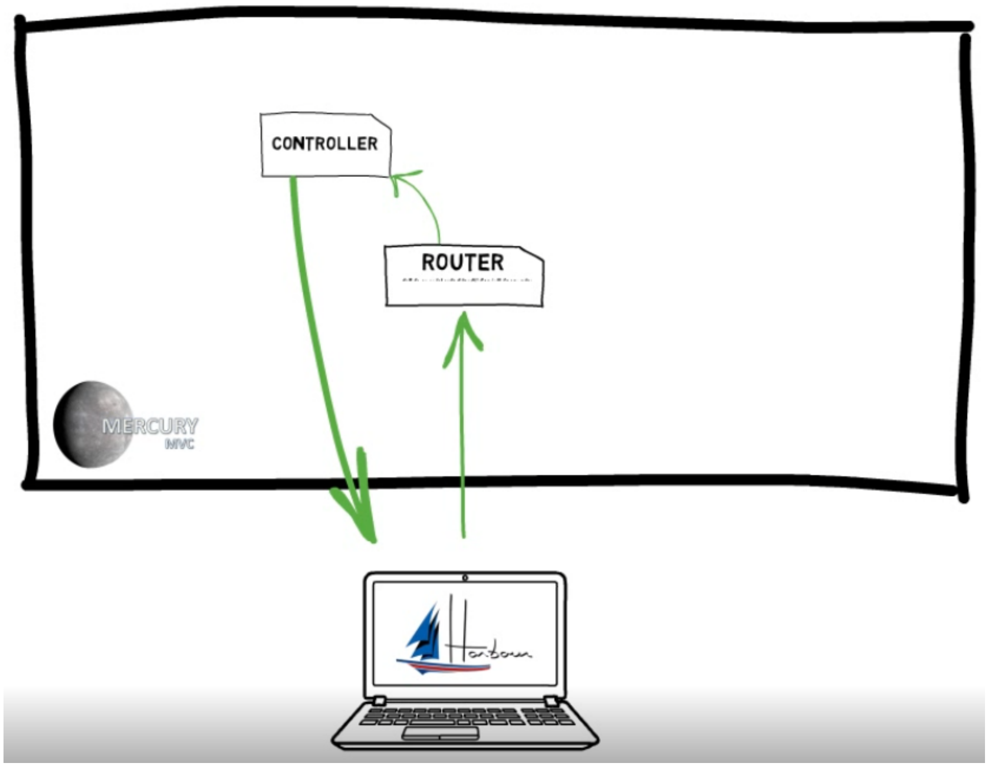


```
{
    name: "John Kocinsky",
    age: 38,
    date: "20011107"
}
```

Summary:

- The system receives a request that is routed to our controller
- The controller processes data and gives a response in json format

So conceptually we would have this scenario → https://youtu.be/VnwM4kQKKmw



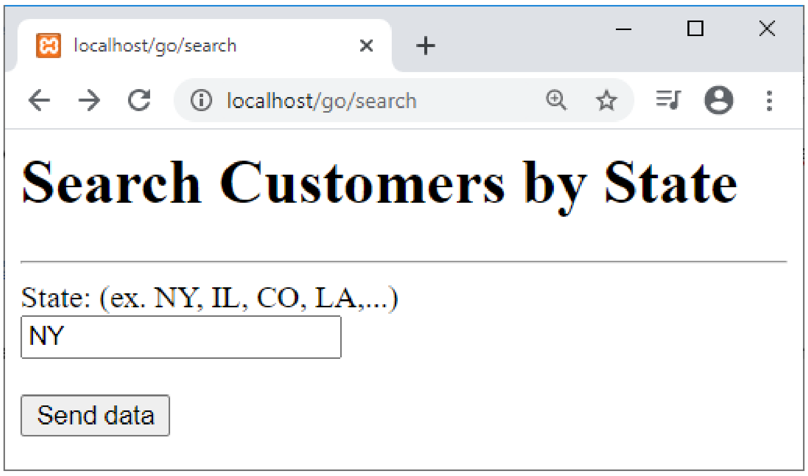
Autor Carles Aubia
Date 10/06/2020
Version 1.02

Creation of a view form to search the state

We are going to create in the search.view a simple form in pure and simple html to request the list of users of a state. Super simple html code might look something like this

```
<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>
```

If we run localhost / go / search the following screen will appear



Search Customers by State

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

NY

Send data

Now perhaps comes some of the concepts that cost more to understand. This form needs to add the "action" where we want the request to be made, that would be what program to execute when we hit the button. The idea is that when we fill in the data and press the button, our GetByState() method will be executed by our customer controller. How do we do this final step?

We will create a path in our index.prg that could be similar to this

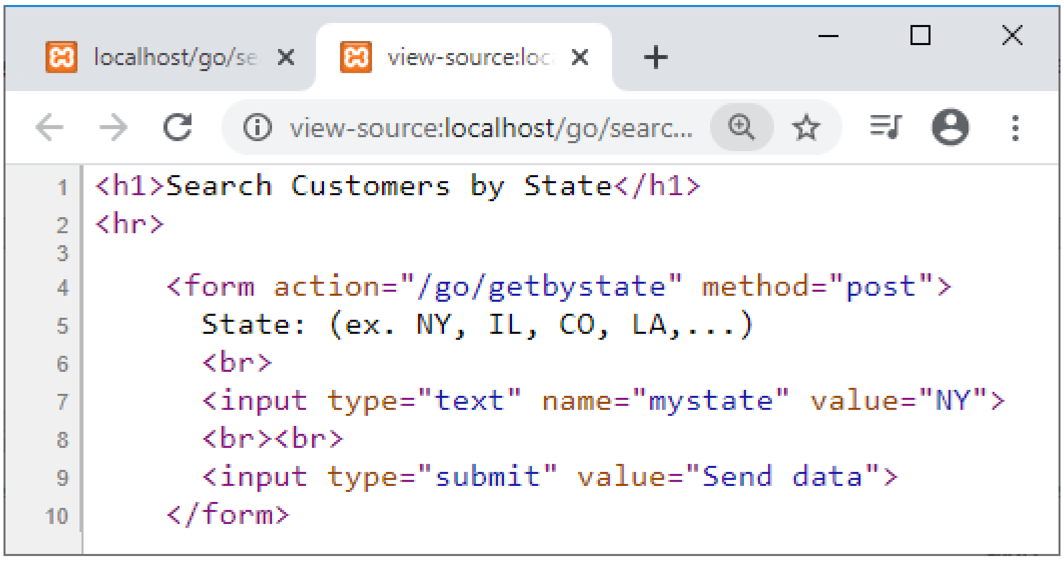
```
DEFINE ROUTE  'getst'  URL 'getbystate'  CONTROLLER 'getbystate@customer.prg'
   METHOD 'POST' OF oApp
```

We should already have assimilated and understand the meaning of the route definition

Now we only have to put in the view search where we want to route it when we press the button. Nothing as easy as adding the following and that is the key to learning how to route our forms correctly

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

What our system will do when creating the form is to replace {{Route ('getst')}} with the route (url) where it should create the request to the server. If you look at the index.prg (Router) we have defined the key "getst" and it points to the controller 'getbystate@customer.prg' so it really (and if we consult the code of the page once loaded, with the right button- > see page code) we will see the real substitution



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

    <form action="/go/getbystate" 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>
```

It is important to understand how when we create our view we have to manage our routes. For this there is the router and we should not worry about paths, urls, ...

Summary:

- We have created a route for our system to show us a form to enter data
  We mount the view so that the form points to / go / getbystate when it is executed

Collection of parameters from the Controller

Now we only have from our GetByState() method to collect the parameter of the "state" that has been entered in the form. To do this, the oController object that reaches all our methods will help us to do this with the methods and objects that it has.

Object oRequest

The oController: oRequest object is in charge of listening to the requests and collecting the parameters on the server. We can use the following methods to collect the parameters

oRequest - Métodos	**Descripción**
Method()	Method by which the request has been made from html
Get(cKey, uDefault, cType)	Retrieve 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()	Retrieve from Url Query
GetUrlFriendly()	
GetCookie(cKey)	Recover Cookie

Basically we will use the Method Post() y el Get() .

Both the Post (), Get (), Request () method have the following definition:

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

<cKey>	Parameter name
<cDefault>	Default value. Default is an empty string
<cFormat>	If we want to format the data. They can be typical of Harbour: 'C', 'N', 'D'

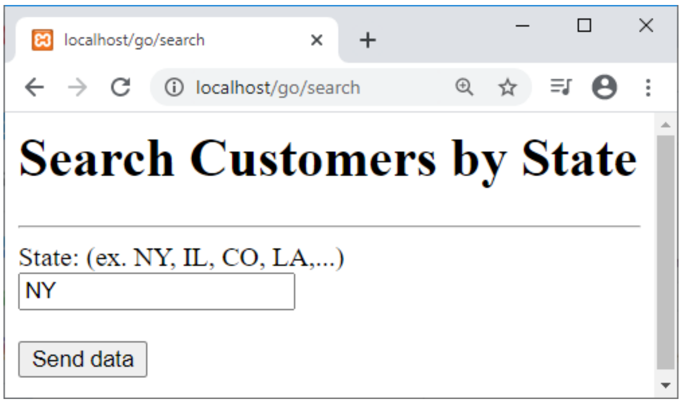
Autor Carles Aubia
Date 10/06/2020
Version 1.02

In the form if we look, the name of the variable is "mystate" and we know that the form is executed by the method "post" we would retrieve the variable as follows.

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

If we do the test and modify the controller to show us the variable that comes to us from the form in this way we can check it. It is a good way to test it.

```
METHOD GetByState( oController ) CLASS Customer

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

       ? 'Estate: ', cState

RETU nil
```

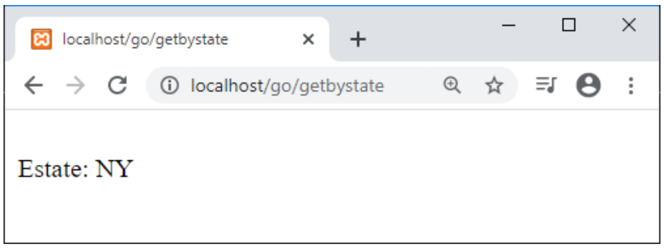

localhost/go/search

Search Customers by State

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

NY

Send data


localhost/go/getbystate

Estate: NY

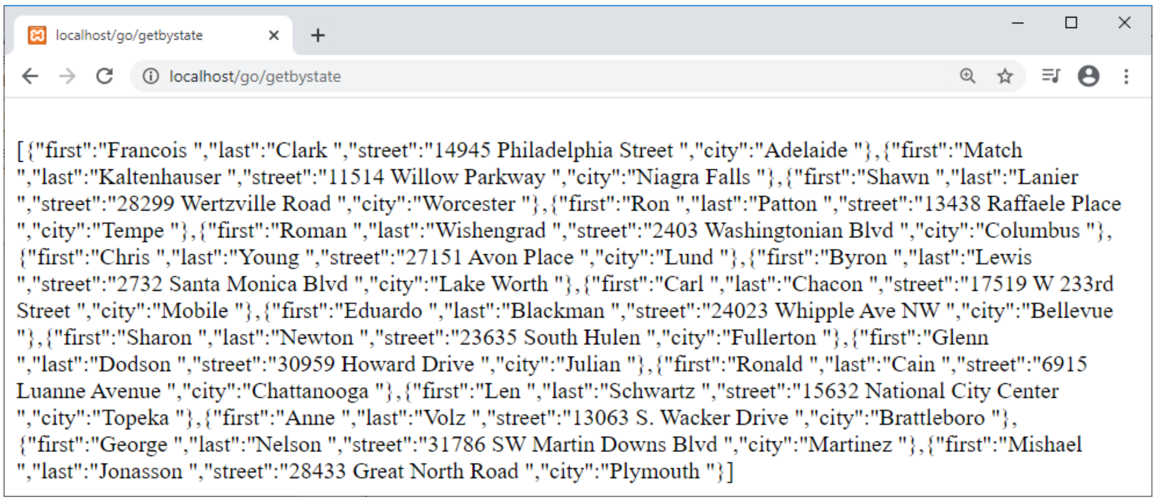
Summary

- Creation of a “search” route that goes to a controller that redirects it to the “search” view.
- The "search" view mounts the form that points to the getbyst method of the controller when you run it
- The GetByState () method we see how it collects the parameter "mystate" and shows it on the screen

Collect parameter and search in table

Now that we know how to recover the parameters, we already have all the pieces of the puzzle. We just have to add the code to search the table for the "state" we have received and put the records in a table. To know if we did it right, at the end we will show the 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
```



```
[{"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 "}]
```

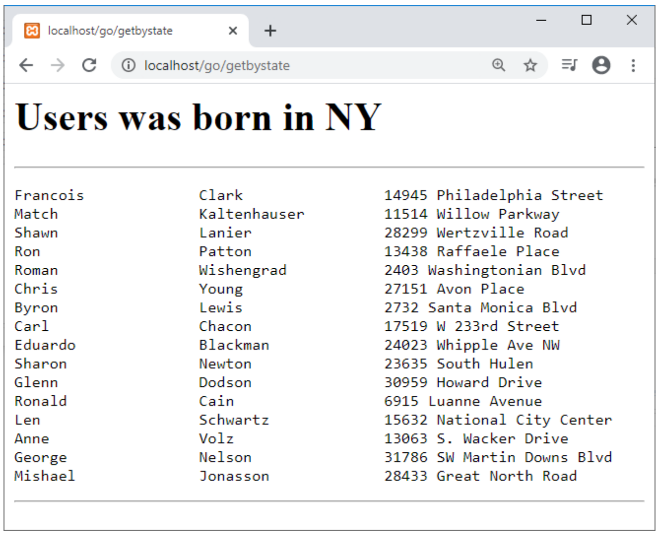
We see how the controller, the person in charge of processing the data has carried out its process correctly. Now the last step would remain. Once the controller finishes its process, what it should do is pass the data to the view so that it does its job: painting the screen.

Pass the result to the view

Once the controller is finished, we will call a view that is responsible for creating the web. We will call it listbystate.view and the simple code could look something like this

```
<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>
```

And the end result would be an output on the screen similar to this



Users was born in NY

```
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
```

Summary

- Creation of a search route that goes to a controller that redirects it to the Search view
- The search view cute the form that points to the controller's getbyst method when you run it
- The GetByState () method we see how it collects the parameter "mystate" and opens the table to load the data
- We call the View listbystate and pass the data to it to paint the screen

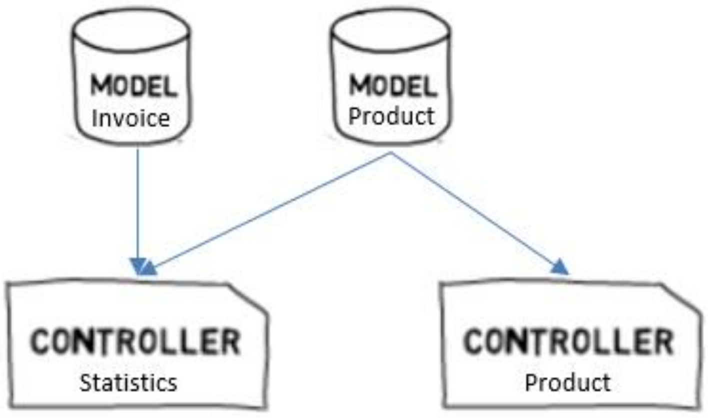
Model

In the same way that we had everything in the view initially, now we have extracted the part of the data process in the controller and we already separated the data process a bit from the view. Can we survive like this? Well surely if, like when we had everything in the view, but we do not have it completely correct, yet ☹

Basically the model is the part that connects to our data and creates its inputs and outputs. The main objective is to provide us with the data that can be a dbf table, such as a mysql bd, Oracle, defined data arrays, …

So we must extract what we have done so far from the controller and encapsulate it in a data model that is responsible for treating the Customer table. In the end, the controller, if you need that model or models, will open them and ask for the data you need to process them later.

From here we have to think that our application as it grows will have several controllers as we have commented previously and each controller will use its / s data models, but there may be controllers that use the same model, so if we create our system, our model is universal for any controller that needs it



Creating the Customer Model

For our exercise we will create a simple data model that will simply open the data table and have a method called RowsByState (cState). We could make it simple this way

```
CLASS CustomerModel

      DATA cAlias

      METHOD New()                        CONSTRUCTOR

      METHOD RowsByState( cState )

ENDCLASS

//----------------------------------------------------------------------------//

METHOD New() CLASS CustomerModel

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

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

Once we have the model created, the change to be made in the Controller series 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
```

And at the end of the controller insert the model

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

We have to think that we when we create a web application, unlike an application e.g. Windows in which we compiled and then linked libraries that we needed in the program, here on the web we will indicate in this way if we need any file and it will be resolved at runtime.

And that's all !!! 😊

With this part we must think that we basically abstract the entire data model, that we share the data model and ... if one day we change the Customer.prg model for another one but instead of accessing dbfs it does so for mysql, we ONLY have to change one on the other without having to touch anything else from our application!

Summary of the processes involved in this consultation:

1- We have created a route in our index.prg to manage "search"

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

2- When the request arrives at the controller, it redirects it to the view "search.view", which creates the form to send to the browser

3- This form that will reach the browser has in the <form> tag the path that it should take when we press the send data button.

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

4- This implies that we must have in our router (index.prg) created a route for "getst" that accepts it for the 'post' method, which is the one we will use in the form

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

5- it will also have the variable that will be sent to the server "mystate"

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

6- In the controller customer we will have a 'getbystate' method that will listen to the request

7- The GetByState () method will retrieve the parameters of the form, in this case 'mystate'

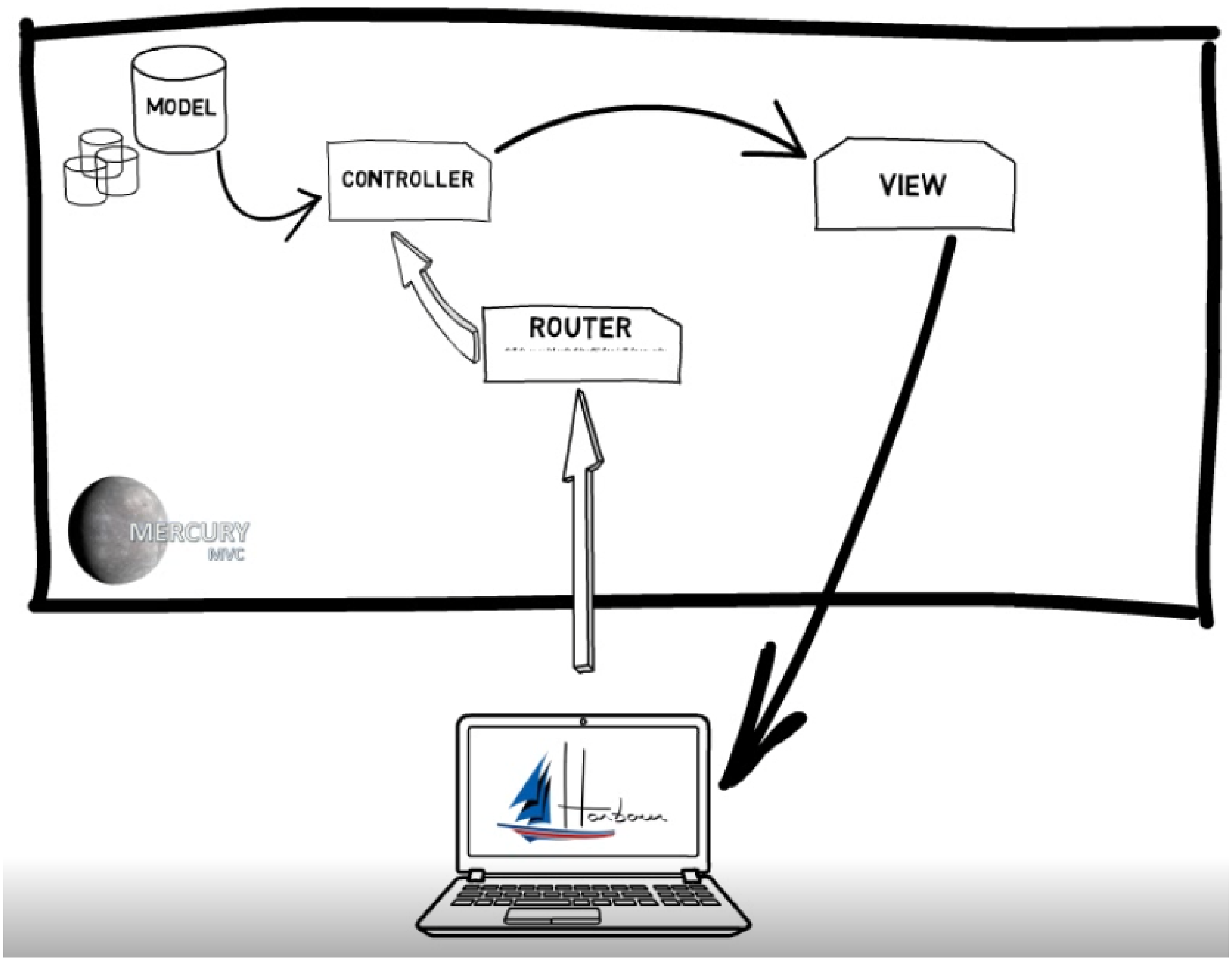
```
local cState := oController:oRequest:Post( 'mystate' )
```

8- We will open an instance of the Customer data model and ask for the "state" data that we have recovered

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

9- We send the data to the "listbystate.view" view for you to paint. This view will retrieve the data and paint it

This is a typical MVC circuit and if we have got this far, our mind should already be able to see this scheme → https://youtu.be/FN9Urv0Ouxs



And if we continue analyzing the scheme we will see that we have circled 2 times:

1- When we request the "search" which is the screen to enter data
2- When we enter the data and request the information

This is the basis of MVC. If we can understand it and apply it, it will give us great security and productivity over time.

Apart from the data folders, lib, the logic of the application that we have created resides in these folders and files.

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

   /index.prg
```

We know quickly what to touch and where to go if we have a problem on the screen, or in data access, or its process ... It really is a very ordered and powerful system that will help us in the design of our applications

Autor Carles Aubia
Date 10/06/2020
Version 1.02

Part 2 - Advancing the pattern

If we have managed to get this far we have the most important part achieved and that is the conception of the entire system. Understand where it enters, leaves, what each module does and what function they have.

But the issue does not end here ☹, we can do much more to make our system efficient, agile, robust, ...

The first law in a backend is ALWAYS to validate the data.

Mercury has a TValidator() class that can help you in that task, it is not mandatory and each one can use their system since it can be controlled in many ways.

One of the most common mistakes is to use javascript plugins to validate data entries and this looks great on the screen when we enter data, but we always have to think about how the system can be altered because there is always someone ... who tries it.

Following the example of the customer search for a "state", the first thing to think about is that, when receiving the parameter, we know that the format must have the following format:

- Characters
- Len 2
- Upper

There is no more, because we have defined it that way. Well this is what we should let pass and if we are not to give an error message.

TValidator

This optional class is very easy to use and will serve us in most cases to be able to control these assumptions

1- We will create a TValidator object

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

2- We will create validation rules that are based on a hash of the variables that we want to validate. Each key will have a string with rules, separated by the character | to separate the different rules, e.g.

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

3- Where we specify that the variable "mystate" must have the following validations:
   - Required – parameter must exist

- Type string, not only numeric
- Exact length of 2 characters

4- We will put in a hash the variables that we want to validate, in this case

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

5- Validating rules → oValidator:Run(hRoles)
6- If the method gives us false (.F.), We can recover the error with oValidator:ErrorString()
7- Send the error to a view that may or may not be the same, eg

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

The view will already have to manage whether or not it receives the 1 parameter and, if it receives it, display it.

8- In the event that the validator passes it, we will follow the flow as before ...

The TValidator () object has a method of "formatting" the variable. This means that we can pass the variable type validation, but we may need to format the variable, e.g. We know that our "state" field must be capitalized. In this case we can use the formatter () method to which we pass a hash with the variables and its format type, e.g.

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

oValidator:Formatter( hData, hFormat )
```

This table shows the different validations that we can control

Rol	Descripción
required	Obligatory field. You need to receive it
numeric	Field must be made up of digits. We must think that we always receive the values in string format, but here we specify that we want them to be numbers, regardless that later we can convert them from string to number
string	Field formed by characters and / or numbers
len:nnn	Field length must be equal to nnn
max:nnn	Maximum field length if we have changed it to numerical format it must be nnn
maxlen:nnn	Maximum length of string field
minlen:nnn	Minimum length of string field

We see how the controller would look:

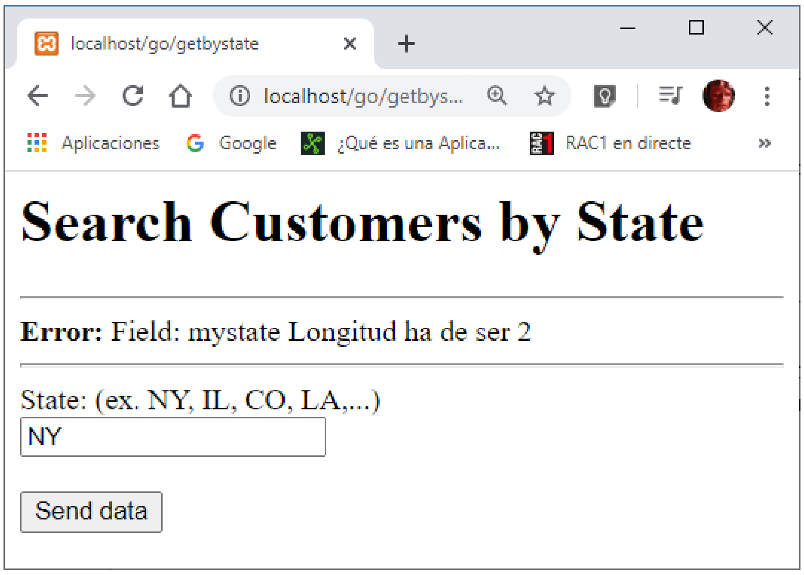
```
METHOD GetByState( oController ) CLASS Customer

  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
```

To finish, note that if you put the validation on, in this case we call the same view and pass the error to it. The view will detect if this parameter is passed and show it.

```
<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,...)
       <br>
       <input type="text" name="mystate" value="NY">
```

```
        <br><br>
        <input type="submit" value="Send data">
    </form>
```

And this is all in this section. Here the important thing is to understand that we must always validate the inputs to our server. If we run this example and enter for example the value "z1aBc", the application will redirect us back to the view and show us the error.



Search Customers by State

Error: Field: mystate Longitud ha de ser 2

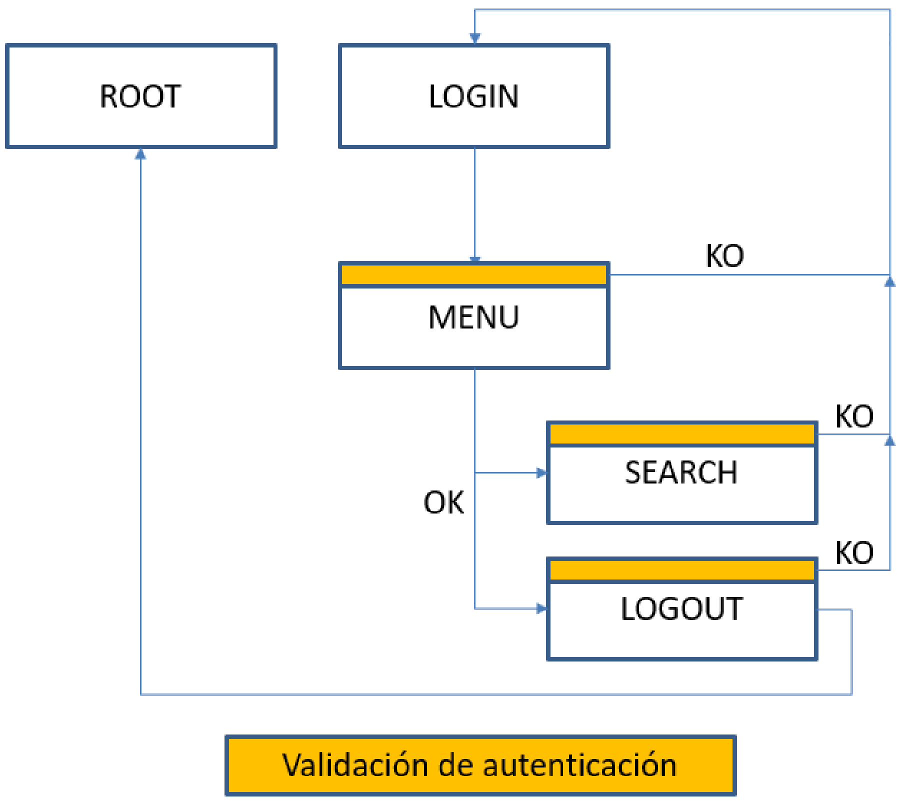
State: (ex. NY, IL, CO, LA,...)

NY

Send data

Autentication

One of the most important issues to consider is controlling access to our system. Surely there will be modules of our application that if you are not authenticated you should not have access. This chapter is about authentication and taking advantage of what we have done we are going to develop this system



We are going to learn how our different controllers can be protected against access if a user is not authenticated. We will make the exercise as simple as possible to easily assimilate the process. In the diagram we see a yellow rectangle that represents the middleware, the logic that checks if it is authenticated or not.

I am not going to comment on the screens that are practically a copy of examples that we have seen, only the news to create the system.

Login

The login system will be just a form, a view. We don't need more.

Router input

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

View: login.view (based on the search.view form)

```
<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>
```

We see that the form is "routed" to "auth" via post

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

Controller → Authorization

We will create the controller access.prg that we will use to validate and log out

Auth()

The purpose of the method is to authenticate the data entered in the login form. The process will be as follows:

- We recover data
- We validate data
- We verify identity (user / psw)
- If KO we send back to Login
- If OK we create validation in our system and redirect to the menu

```
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' ] }


            //      Iniciamos nuestro sistema de Validación del sistema basado en JWT

                    oController:oMiddleware:SetAutenticationJWT( 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
```

We will comment from the identity verification. As you can see we do it by comparing the string and voila, but when you practice you can create a user model to search and check the existence and the psw.

The idea is that if the validation is OK, we will create a TOKEN that will identify the user every time she makes a request to the server. We can add additional data of ours to the token that we can retrieve every time a request is made and this is what we do in the example we create a has with a pair of data and later we generate the token with:

```
       oController:oMiddleware:SetAutenticationJWT( hToken )
```

Then we will redirect the system to the menu

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

As you can see, we route to "welcome" so we must make an entry in the router. Welcome will be our menu.

Logout()

It will be the second method of the controller access.prg and it will serve to remove the token that identifies the user when closing the system. The actions to be taken are

- Close token
- Redirect to initial page ('root')

```
METHOD Logout( oController ) CLASS Access

  LOCAL oMiddleware   := oController:oMiddleware

  oMiddleware:CloseJWT()

  oController:Redirect( Route( 'root' ) )

RETU nil
```

To perform this action we will do it with the oMiddleware object and its CloseJWT () method and ready. So far we already have access control to our application, it is not complicated.

(Optionally) one last case is missing in the validation topic. Mercury by default assigns a cookie name to the program along with a password that encrypts the token. If you want to personalize your cookie and password, in the file index.prg when the App is defined, it must be specified in this way:

```
  //   Define App

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

We are going to create a menu and we will base ourselves on the example of the start of a view that will jump to another using the 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>
```

Simple code, only 2 links to:

- Search (which is the example we have previously made of listing by state)
- Logout (What to do to disconnect the authentication)

In the router we define welcome

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

Remember that the “welcome” route is used in the authentication

The logout method is already defined in the Access controller, so we only have to register it in the router so that we can route it from welcome

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

Middleware

And the final point: Protect those modules that we want not to be accessed if it is not authenticated, in this case we want to prohibit access to welcome and search. What you have to do is insert in each controller in the new method the following:

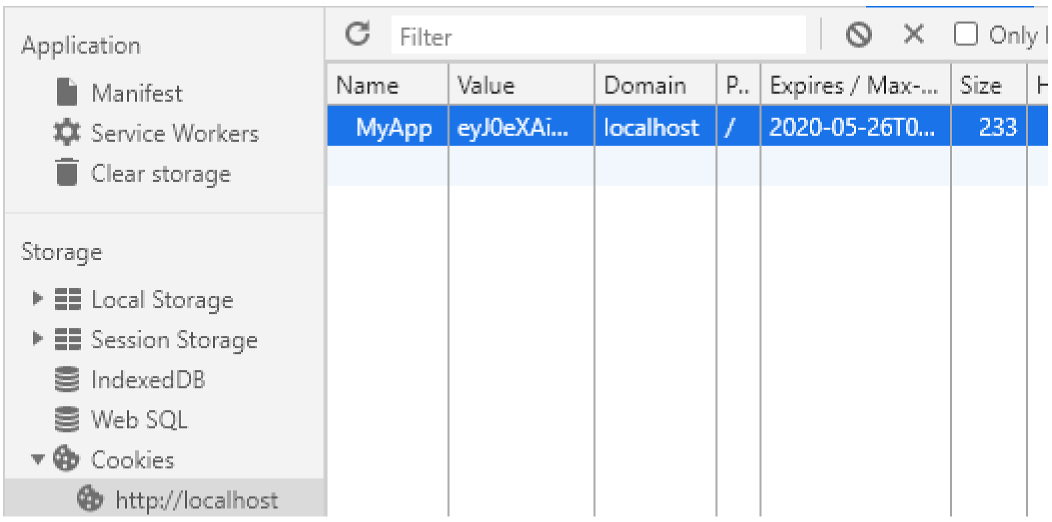
```
#include {% MercuryInclude( 'lib/mercury' ) %}

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

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

What does this command actually do? When trying to access the controller, verify that it is authenticated and if it is not, we will direct it to the login screen

Whenever we use preprocessed commands like these, remember to put the Mercury include at the start of the program

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

As a curiosity, when you access the login and authenticate yourself, if you look in the inspector, in the application section, the cookies will see how the cookie that carries the security token has been created for you. If you delete it or log out, you will no longer be able to access the protected modules and you must authenticate again



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

And with this point we finish the practical exercise. Now you only have to scale your needs in an easy and orderly way 😊

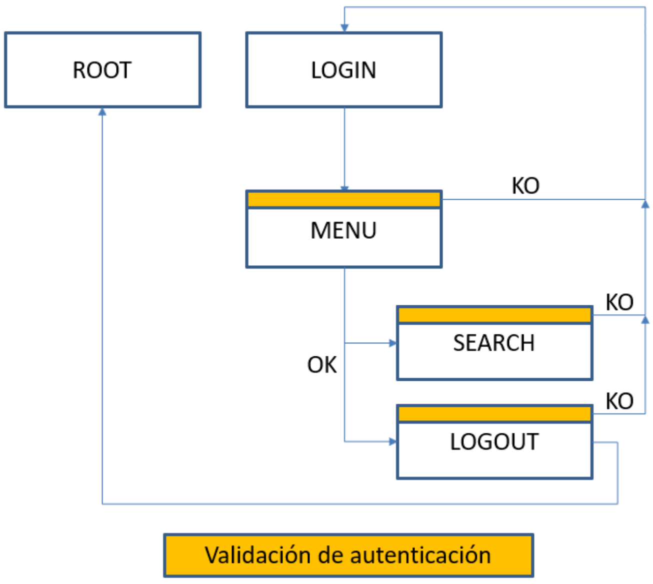
Autor Carles Aubia
Date 10/06/2020
Version 1.02

Final summary of our system

We have already finished our first system.

- Home screen → localhost/go/
- Authentication system → localhost/go/login
- Menu with options → localhost/welcome
- Search module → localhost/go/search
- Modules with authentication validation system

Note that if we try to access the search or welcome module without being authenticated, the system will automatically redirect us to the login screen. Perhaps now our initial approach will be seen and understood more



Made one, made all !!! and in this way we can protect our modules from unwanted access.

Autor	Carles Aubia
Date	10/06/2020
Version	1.02

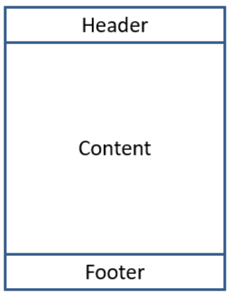
Part 3 - Advanced techniques

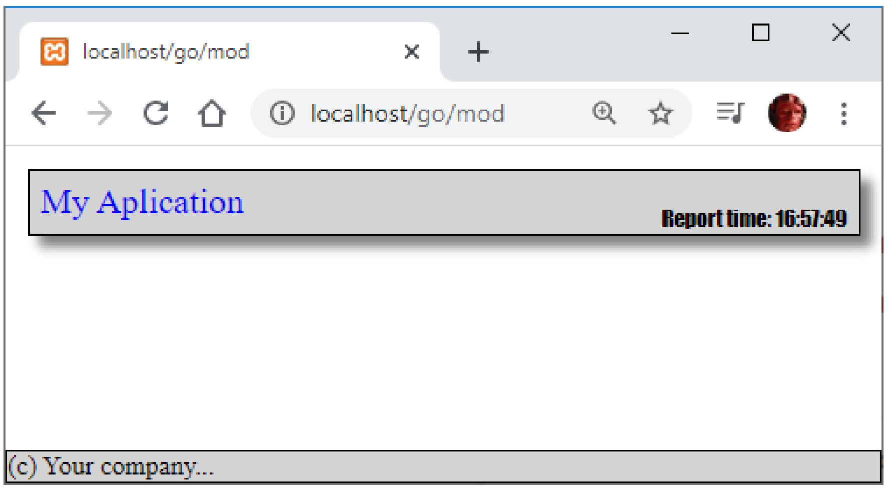
The reason for this 3rd part is to consolidate the system and add a series of functionalities that will make life easier for us when programming our application..

View() - View de View

We have seen at a very conceptual level how to create a view, hyper simple, but the part of the screen painting is one of the most headaches. Here we must also have the virtue of the MVC that will help us optimize code.

Let's imagine the typical page that has its header, body and footer (header, content, footer). Practically the header and footer of the page will always be the same or most of the time it will be.





My Aplication

Report time: 16:57:49

(c) Your company...

What it is about is to divide into different files and use them when needed. We are going to set up an example creating a file that will paint the header and another one will ask for a page

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>
```

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

The idea is that from any view we can load this header and this footer using the View function (<cView>) using {{...}}

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

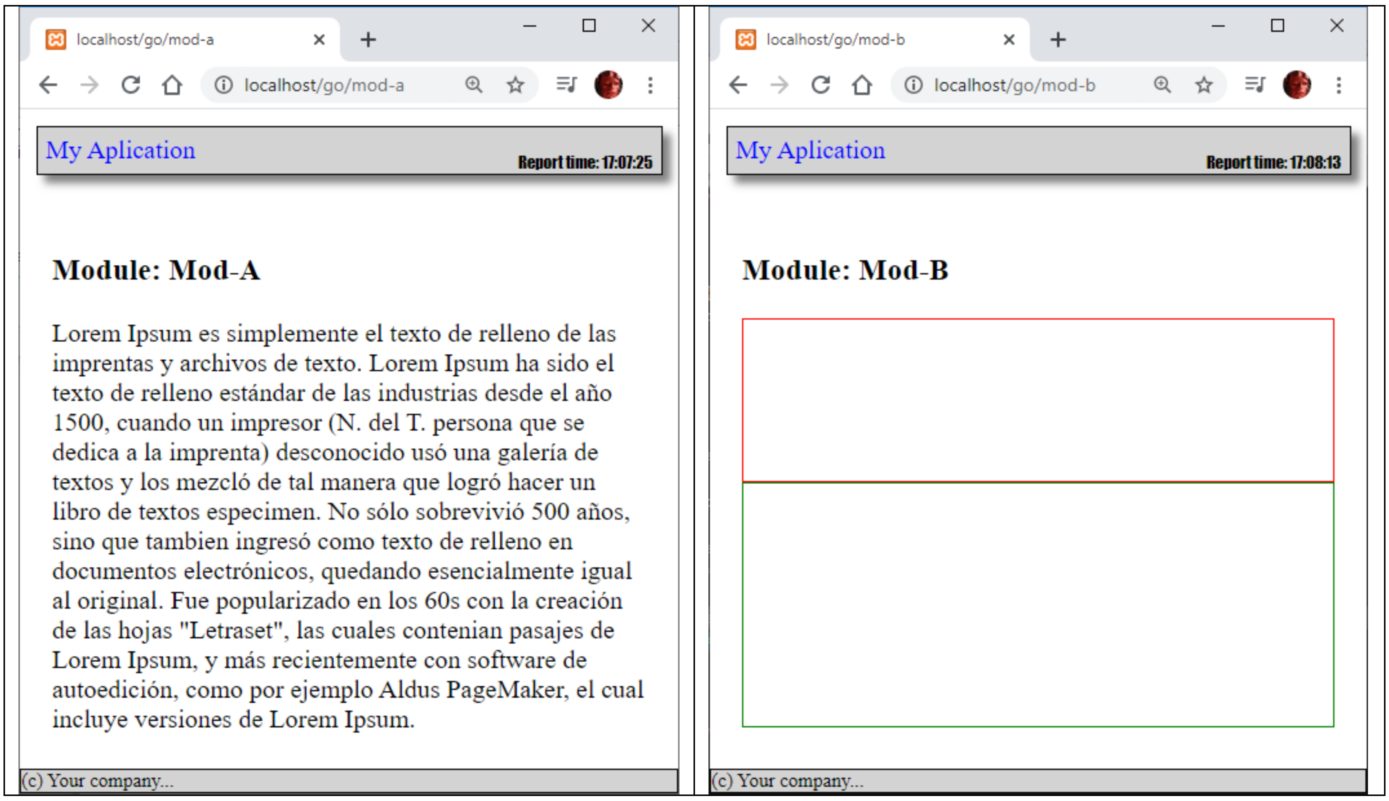
The code would be simplified later when creating a page in this way

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>

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

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

</div>

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


localhost/go/mod-a

My Aplication Report time: 17:07:25

Module: Mod-A

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. Fue popularizado en los 60s con la creación de las hojas "Letraset", las cuales contenian pasajes de Lorem Ipsum, y más recientemente con software de autoedición, como por ejemplo Aldus PageMaker, el cual incluye versiones de Lorem Ipsum.

(c) Your company...

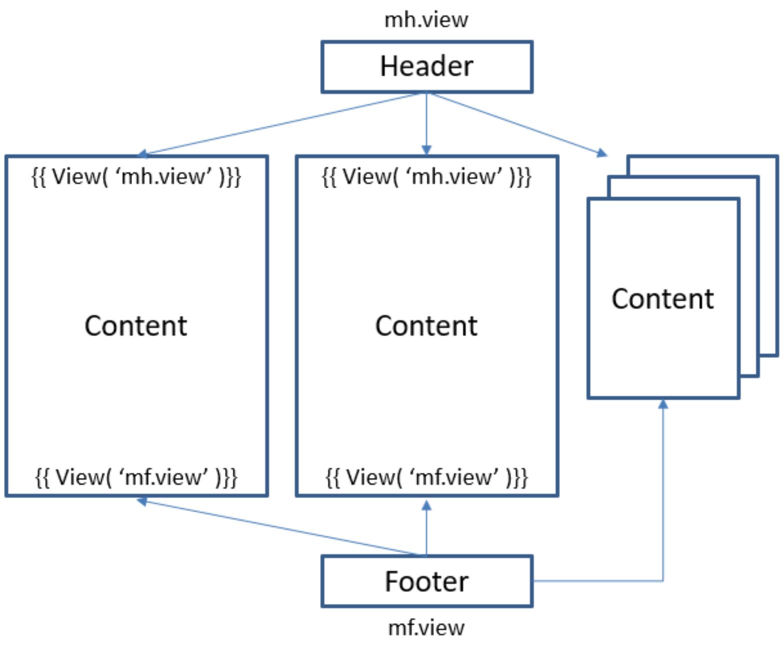
localhost/go/mod-b

My Aplication Report time: 17:08:13

Module: Mod-B

(c) Your company...

In this way, the result can be seen to be the same with the difference that maintenance is different. Let's imagine we have 10 screens designed and each one with all the code. If we want to modify the header for all, we will have to modify all the views one by one, while in this way modifying only the header, it will automatically be reflected in all the views where it is used.



Css()

And to finalize the optimization it would be necessary to speak of the css. More and more we will use the css because we will be assimilating it and we will see its benefits when it comes to layout, but... do we keep putting the css code inside the view itself? Good practice advises having it in a separate file and we will follow the advice ...

We will create in our project a folder called css and inside we will create in our case a file that we will call app.css. In the for example we will put all the css that we have used in these last examples, for example if we see the myhead.view example we could put in the app.css all its used styles remaining like this

app.css

```
/*     Header */

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;
}

/*     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;
}
```

And it only remains to insert this style file wherever we can use it using the function Css()

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

To finish the myheader and myfooter code it would look like this

myheader.view

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

myfooter.view

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

And finally on the page that we assemble, we will load the css and the header and footer views, with which the code will be highly optimized

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' ) }}
```

Layout

Once the entire process has been completed to learn how MVC works, we will see how layout fits into our projects. What we have seen so far is the logic of our entire application as all the pieces fit together: Router, controller, model, view, middleware, validator, ...

Layout is the process in which we leave our application "pretty" 😊. It is not the objective of this manual to teach how it works, there are already hundreds of manuals on the net that you can see its foundations, but it does represent a small change on some screen at this point.

We will choose the Login screen to "lay it out" using Bootstrap to understand this final process.

The original view was this

```
<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>
```

For the change of look, we will create a view that will load the necessary Bootstrap libraries that we will call 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">
 <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>
```

And taking login.view as a base, I modify it leaving it this way and everything that is used has been seen throughout the manual:

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

I have added a couple of images that I have put in the / images folder that will be part of our project

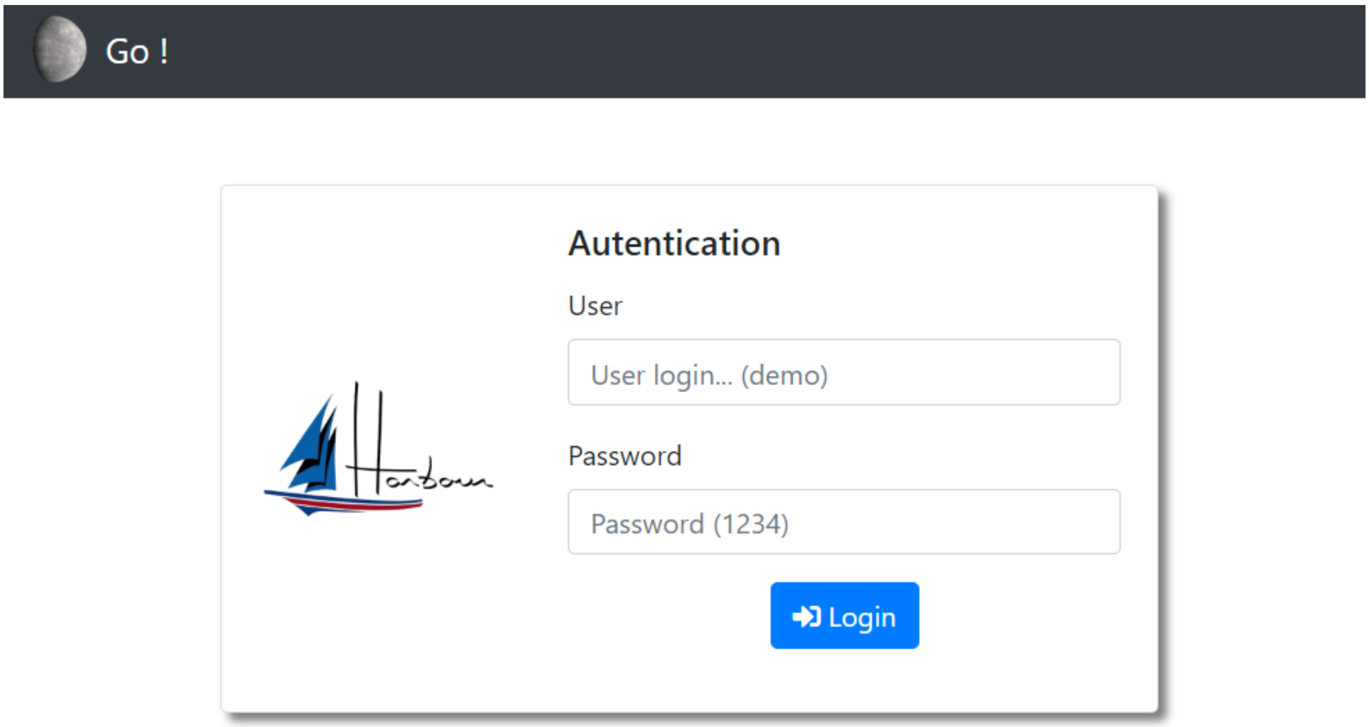
```
{{ View( 'head.view' ) }}
{{ Css( 'login.css' ) }}

<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <img src="{{ AppUrlImg() + 'mercury_mini.png' }}" alt="Logo" style="width:35px;">
  <a class="navbar-brand" href="{{ Route( 'root' )}}">&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;">
```

```
  <div class="row no-gutters">
    <div class="col-md-4 container_logo">
   <img src="{{ AppUrlImg() + 'harbour.png' }}" class="rounded logo" alt="Cinque Terre">
    </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>
</div>

</form>
```

And I have created a small css file → login.css with this content

```
.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%;
}
```

So far we must already know and fit these pieces. The result is as follows



Nice ? Well in this case part of the merit is from Bootstrap → https://getbootstrap.com/, but the conclusion when arriving here is that ALL the application logic continues to work, no controller, model, or router has been modified, validator,... we have ONLY changed the view that we had hyper simple with a bit of bootstrap, that's all and this is part of the magic of the MVC when understood. Each piece does its job, independently of the other. Bootstrap has helped us make it beautiful, but it has only "painted". We have built the whole process behind and moves the system. We must separate all these concepts.

If someday I want to use other frameworks to lay out such as materialize → https://materializecss.com/, I just take this login view and tweak it.

All the rest of the application will work the same !!! 😊

WebServices

Reaching this part of the manual, we are left to talk about how we can make a webservice and it will be a very easy task with everything we have assimilated.

We will rely on RESTful webservices simply because they are the easiest to manage, most powerful and because the last few years have emerged as the most popular and used.

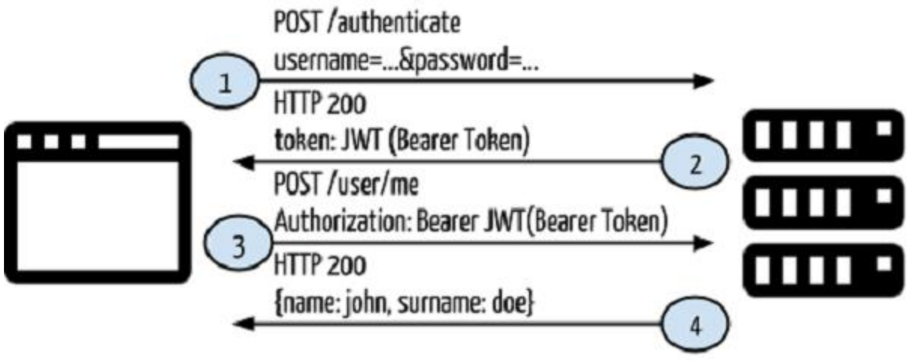
Basically a webservice is as the word indicates, a service that gives you a website, we could focus on queries at the moment. We will connect to a ws for example to check the time, the weather, ... up to corporate ws or that an authentication is required to check data, let's say more sensitive.

Basic operation of the WS

Basically the operation of a ws system is based on tokens. We are going to reproduce the same system using tokens and basing the OAuth 2.0 authorization standard and it is of type "Bearer", we will see later.

When we connect to the server that provides us with ws the flow is as follows:

- We normally authenticate with user/psw
- The server if Ok returns a Token
- Each request we make will be attached in the header: Authorization: Bearer JWT
- Each request we make will be attached in the header


POST /authenticate
username=...&password=...
1
HTTP 200
token: JWT (Bearer Token)
2
POST /user/me
Authorization: Bearer JWT(Bearer Token)
3
HTTP 200
{name: john, surname: doe}
4

-

Method GetByState() – Customer list by state

We return to focus on the example we have made with "search" that we were looking for clients of a certain "state". The process is already summarized in the following way:

- Collection of parameters
- Validation of parameters
- Data processing with the model
- Response passing the data to a view

The WS is practically the same! It only changes that before we sent the error or the result to a view and now we will send it directly to the browser in a specific format, in this case in json format.

The most used format for its flexibility, size, etc, ... is json so we will show how to do it.

We will create a new controller that we will call ws and we will create the same method that we had in the previous one, GetByState (). The result would be this

```
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
```

If we compare the code from GetByState@customer.prg with GetByState@ws.prg only the type of response given by the controller changes. On the one hand it sends to the view and on the other it responds directly by sending the data in json format. To respond in json format we will use the method of the oReponse object: SendJson () as explained above.

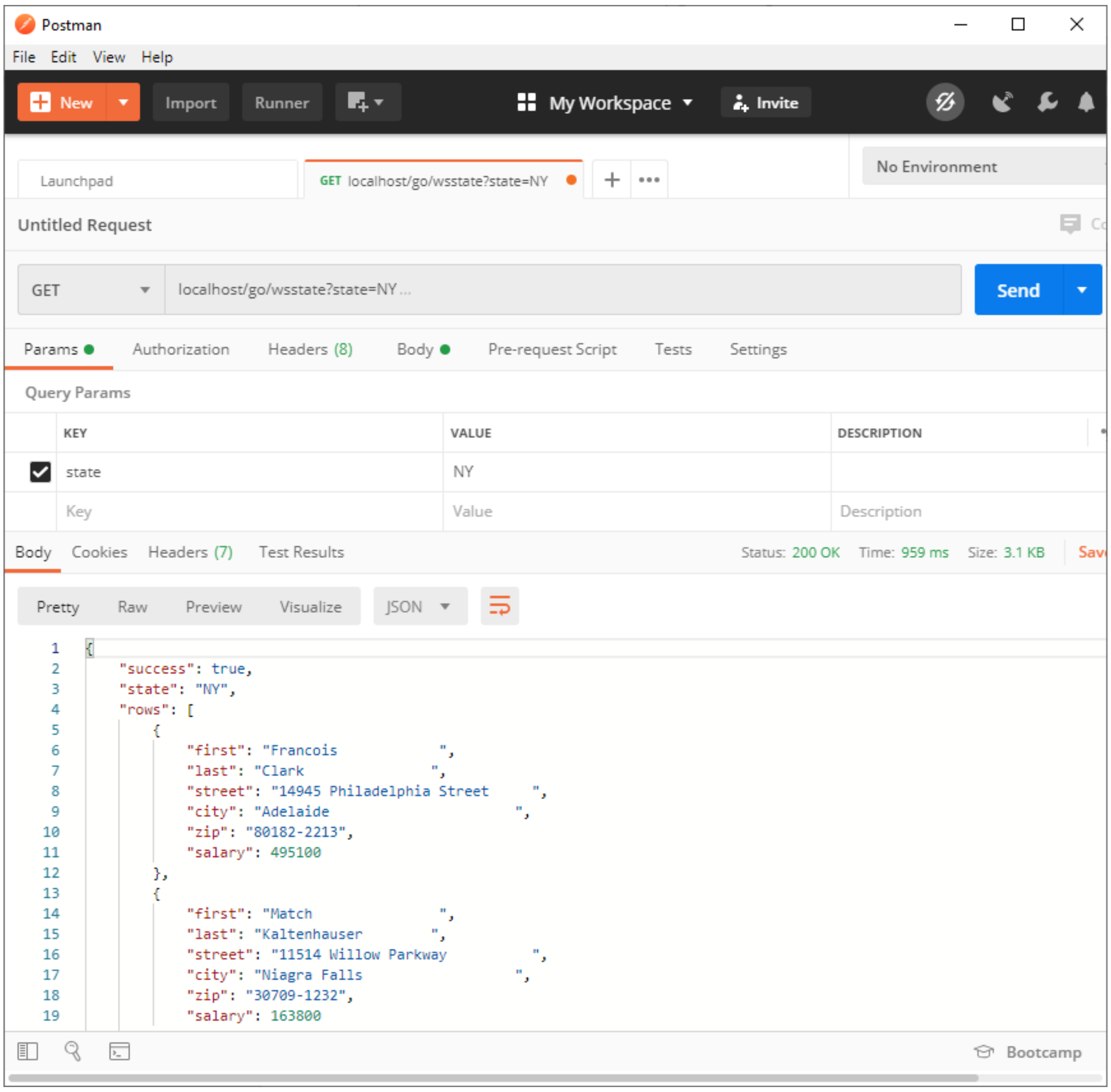
I also choose a type of response that I like, the json will have this format::

- success (true/false)
- error (case success = .f.)
- rows (process result)

We know how our module works, because we only have to create its routing in the index.prg

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

We see that the url we choose is 'wsstate'. Now we only have to test it with one of the many programs that exist. I choose postman to take the test.



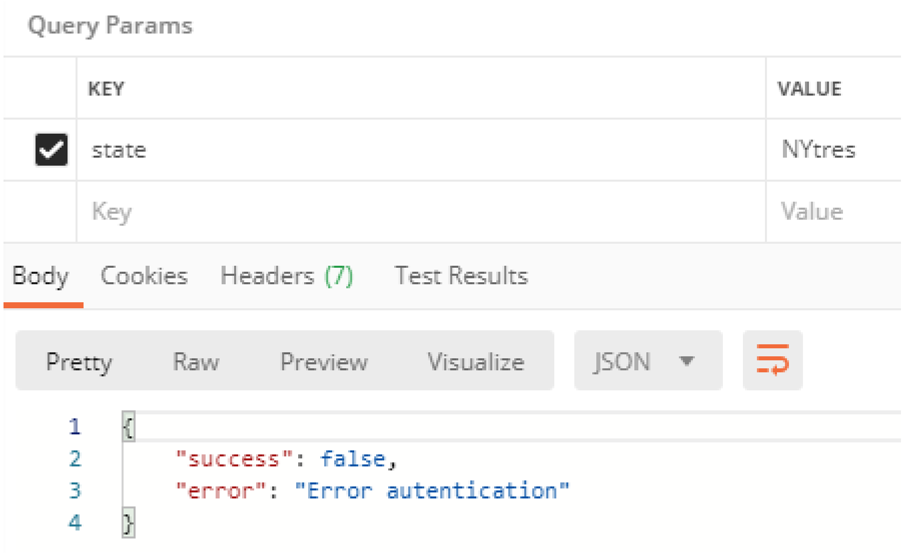
GET localhost/go/wsstate?state=NY

KEY	VALUE	DESCRIPTION
✓ state	NY	
Key	Value	Description

Status: 200 OK Time: 959 ms Size: 3.1 KB

```
{
    "success": true,
    "state": "NY",
    "rows": [
        {
            "first": "Francois            ",
            "last": "Clark              ",
            "street": "14945 Philadelphia Street     ",
            "city": "Adelaide                    ",
            "zip": "80182-2213",
            "salary": 495100
        },
        {
            "first": "Match               ",
            "last": "Kaltenhauser       ",
            "street": "11514 Willow Parkway          ",
            "city": "Niagra Falls                ",
            "zip": "30709-1232",
            "salary": 163800
```

We can check that the system answers us and gives us the correct answer in json. In the case of putting a wrong entry the system responds the following



	KEY	VALUE
✓	state	NYtres
	Key	Value

```
{
    "success": false,
    "error": "Error autentication"
}
```

We already have the first part of the ws prepared, now we lack the lack of authentication in the event that we want to protect our ws and only let it be used by whom we authorize.

Method Auth() - Creation of authentication system

We will be based on the Auth method that we created in the controller Access.prg because the concept is the same. One of the things to keep in mind is that we will always return the answers in json format. Following this pattern, we will define our system with a structure, in which we define the key success to know whether good or bad and the other information keys. In this case for authentication, we will return:

- Success (true/false)
- Error (case success = .f.)
- Token (case 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
```

```
//    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 autentication'

            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:SetAutenticationToken( hToken, 120 )

            //     Preparamos respuesta

                   hResponse := { 'success' => .t., 'token' => cToken }

      else

            hResponse := { 'success' => .f., 'error' => 'Error autentication.' }


      endif

      oController:oResponse:SendJson( hResponse )


RETURN nil
```

Only 2 things change:

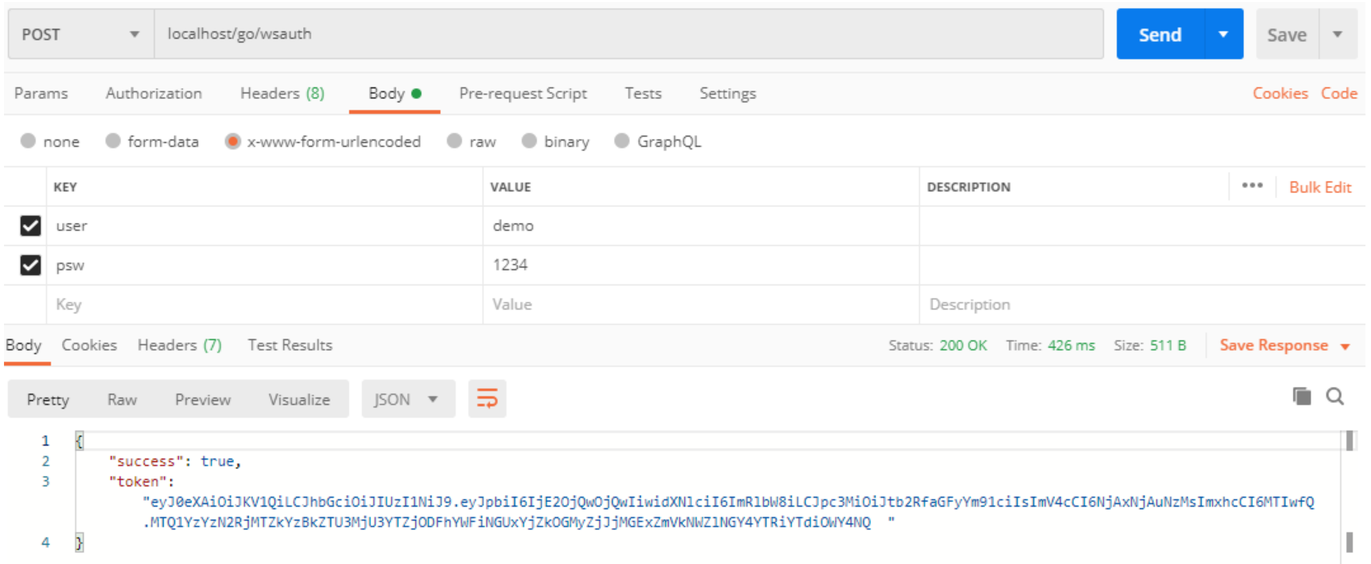
- We generate the token with the method → oController:oMiddleware:SetAutenticationToken()
- The answer is via json → oController:oResponse:SendJson(hResponse)

We will create the following entry in the router

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

And we already have it ready to test with the Postman. We will have to take into account:

- Access via POST, so we have decided on the router
- We must pass the parameters user=demo/psw=1234



	KEY	VALUE	DESCRIPTION
✓	user	demo	
✓	psw	1234	
	Key	Value	Description

```
{
    "success": true,
    "token":
        "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpbiI6IjE2OjQwOjQwIiwidXNlciI6ImRlbW8iLCJpc3MiOiJtb2RfaGFyYm91ciIsImV4cCI6NjAxNjAuNzMsImxhcCI6MTIwfQ
        .MTQ1YzYzN2RjMTZkYzBkZTU3MjU3YTZjODFhYWFiNGUxYjZkOGMyZjJjMGExZmVkNWZlNGY4YTRiYTdiOWY4NQ  "
}
```

As we can see, the server returns a success = true and a token

This token is the one that we will use in each header of each request that we will make each time to our server.

Method New() – Token Validation

Before we have to protect our ws so that you work with an authentication system. We will follow the same system that we described before, but first a nuance to consider. Our WS that we have defined has 2 methods:

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

The Auth() system is a public method that does not need authentication. We can always access to request a token, while GetByState () will be a private method that previously needs authentication. How should we work this situation?

As we saw in the authentication topic we used this preprocessing to authenticate

```
AUTENTICATE CONTROLLER oController ERROR VIEW 'login'
```

Time the system is the same but with a modification:

- We must indicate that, in case of error, instead of routing it to a view, return a json response
- Indicate that if the method requested is auth () (token request), it is not necessary to authenticate.

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

This is the only difference. In case of error, the system sends a json with the following content

{ 'success' => .f., 'error' => 'Error autentication' }

We can redefine the error message by adding it after ERROR JSON

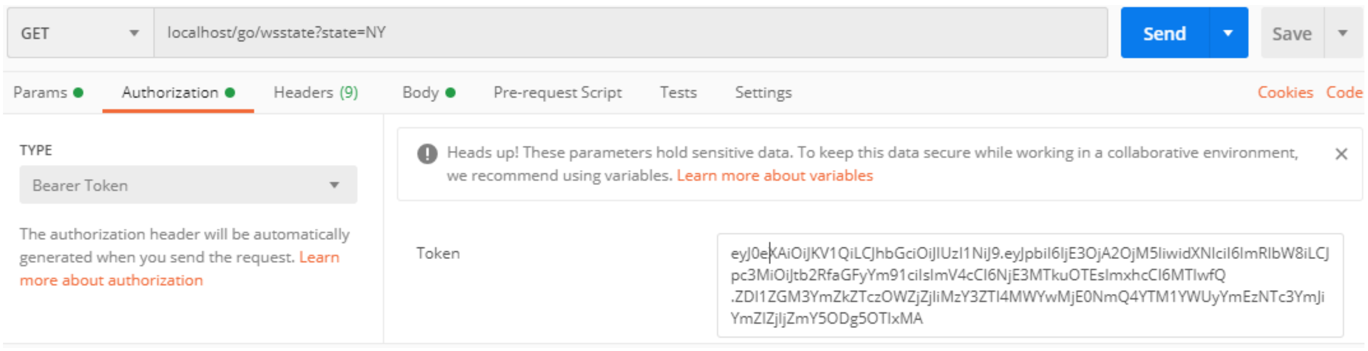
```
AUTENTICATE CONTROLLER oController VIA 'token'  EXCEPTION 'auth'  ;
     ERROR JSON { 'mierror' => 'No autorizado' }
```

Autor	Carles Aubia
Date	10/06/2020
Version	1.02

Podemos redefinir el mensaje de error añadiéndolo después de ERROR JSON:

- Token that they have given us before to authenticate us
- Parameter state that the ws needs to know what state we want
- Request via GET

In the postman remember to specify token



GET localhost/go/wsstate?state=NY

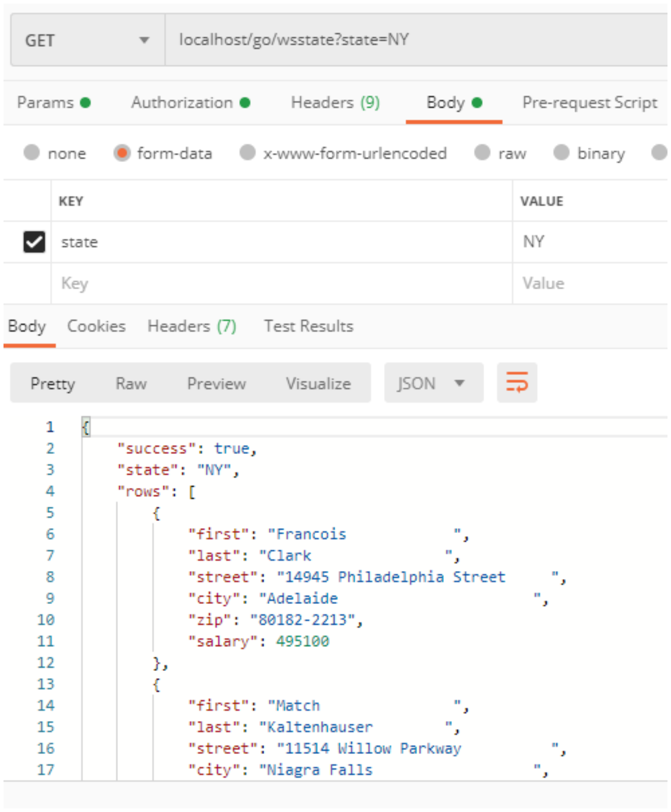
TYPE
Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about authorization

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables

Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpbiI6IjE3OjA2OjM5IiwidXNlciI6ImRlbW8iLCJpc3MiOiJtb2RfaGFyYm91ciIsImV4cCI6NjE3MTkuOTEsImxhcCI6MTIwfQ.ZDI1ZGM3YmZkZTczOWZjZjliMzY3ZTI4MWYwMjE0NmQ4YTM1YWUyYmEzNTc3YmJiYmZlZjljZmY5ODg5OTIxMA

Y los parámetros



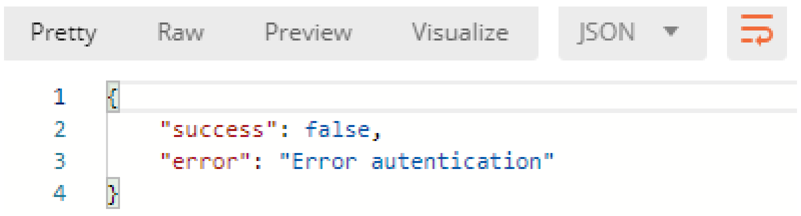
GET localhost/go/wsstate?state=NY

	KEY	VALUE
✓	state	NY
	Key	Value

```
{
    "success": true,
    "state": "NY",
    "rows": [
        {
            "first": "Francois            ",
            "last": "Clark               ",
            "street": "14945 Philadelphia Street     ",
            "city": "Adelaide                      ",
            "zip": "80182-2213",
            "salary": 495100
        },
        {
            "first": "Match               ",
            "last": "Kaltenhauser        ",
            "street": "11514 Willow Parkway          ",
            "city": "Niagra Falls                  ",
```

And we already have our first WS with an authentication system prepared and ready to hang it on our internet server 😊

Remember that when we have assembled the example, when creating the token we have given it a time validity of 120 sec. This means that after this lapse of time the system should give us an error.



```
{
    "success": false,
    "error": "Error autentication"
}
```

Method Data() – View data embedded in the token

To end the chapter dedicated to the WS we can see the data of the token at any time, remember that we had embedded additional data when creating it as input time and user, which are optional and obviously we can or cannot put what we want.

There is a way to retrieve this information in case we want to reuse it once the request is authenticated and is using the method

local hData := oController:oMiddleware:GetDataToken()

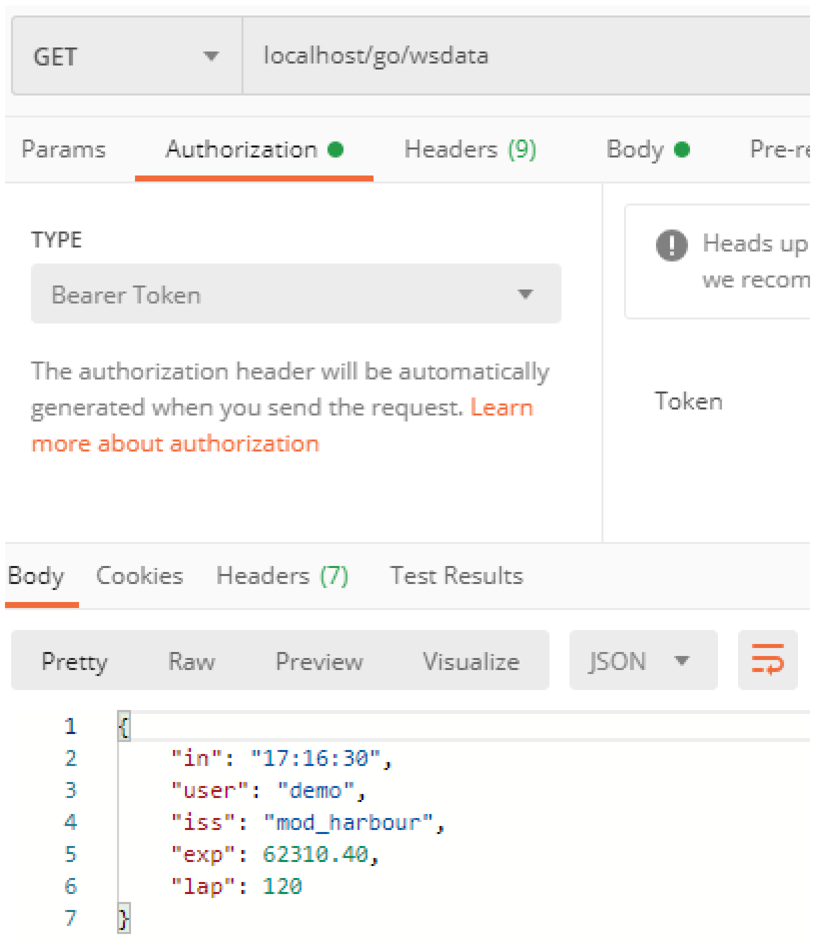
Well, as easy as creating a ws method that we will call to show us this information. We will create the following method:

```
METHOD Data( oController ) CLASS WS

    local hData := oController:oMiddleware:GetDataToken()

    oController:oResponse:SendJson( hData )

RETU nil
```

We collect the token values and return them via json, easier than possible. Now we only have to define the routing

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

And we only need to test it with Postman, remember that the token is valid.



localhost/go/wsdata

TYPE
Bearer Token

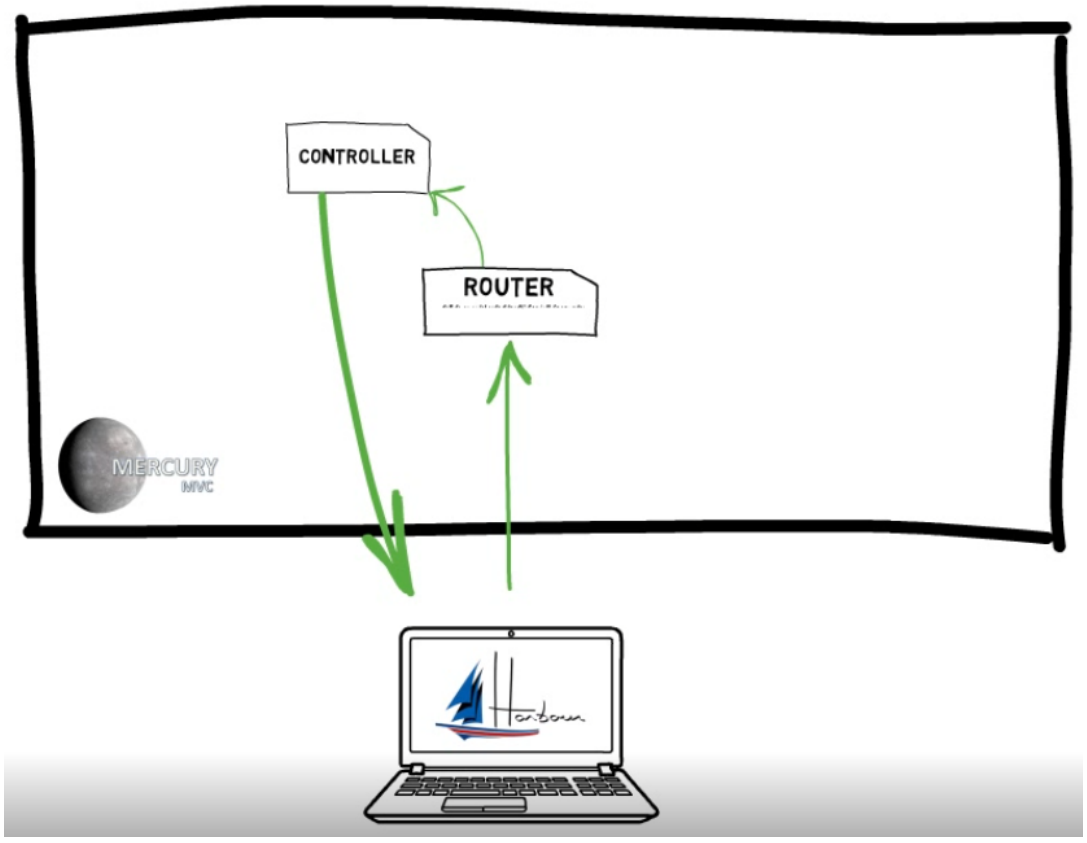
The authorization header will be automatically generated when you send the request. Learn more about authorization

Token

```
{
    "in": "17:16:30",
    "user": "demo",
    "iss": "mod_harbour",
    "exp": 62310.40,
    "lap": 120
}
```

We can check the 2 parameters we define (in, user) plus 3 others that are for internal use by the system. We already have everything necessary to create as many WS as we want

Conceptually we would have this scenario that we already explained in the controller section and that we must now fully understand → https://youtu.be/VnwM4kQKKmw



Conclusions

The Web is a monster that eats you when you want to dominate it. It is necessary to have a discipline when programming and follow a good pattern like the MVC if we want to dedicate ourselves to making serious and professional applications.

At first, like everything, it is something new, perhaps very messy, and sometimes it is difficult to understand, but as we learn and test the system we will realize that it is the way. As we are adding modules to our system, we will see how the proposed structure holds perfectly, there are good foundations and we know where to go to look for everything.

I hope that at this point you have been able to fit all the pieces so that you can get an idea of how to focus your applications. It seems almost crazy for many of us who come from programming environments like Windows, everything we have to touch to create a website (this is just the beginning 😉), but as we understand these techniques it will help us in our projects and we will appreciate their benefits.

Mercury is a Harbour built framework that works like most MVC frameworks currently used with the most popular languages

Harbour is on the Web!

We are here to stay …

"Programming is easy, making programs is difficult"