xaWeb

Fast and easy web development

By Ignacio Ortiz de Zúñiga

Programming guide

xaWeb is a new development platform created by OZ Software that enables management applications for Web environments and Windows (32-bit) and Linux (64-bit) operating systems. xaWeb is based on CGI technology for the creation of dynamic content web pages. CGI technology has important advantages over other script-based development environments such as PHP, such as:

- Speed, as this is compiled code
- Security, since it prevents the execution of scripts that a hacker may have uploaded to our server
- Safeguarding the developer's intellectual property rights by not having to host any source code on the web server, only the executable.

This document does not claim to provide a complete overview of everything xaWeb is and does, nor does it explain how CGI technology works. It is only intended to explain the fundamentals of the tool and in broad strokes its operating system so that any programmer understands what is really happening underneath. Understanding how software works helps a lot to make good software and solve daily programming problems in the right way. **It is important that you read this document before you start working with xaWeb. The learning process will be much faster and with fewer errors**.

OBJECTIVES

xaWeb is mainly aimed at those **Harbour** developers who need to solve important areas of their web developments, without giving up the use of their preferred language. The basic framework that is included with xaWeb aims to cover the needs of any business management software developer, without having to have a great knowledge of HTML, JavaScript or CSS. We consider xaWeb to be a distinctly 'full-stack' tool aimed at small development teams and individual developers.

The main goal is for any Harbour user to start being productive from day one with xaWeb and not see major programming differences between the desktop environment and xaWeb. This was a major challenge since the Web programming and the desktop development environment are very little similar. The first step was to create a complete hierarchy of classes, but one that would hide the complexity of HTML and JavaScript. There are specific controls in xaWeb for each of the existing

controls in HTML and the process of interpreting and producing the HTML code is done automatically by xaWeb.

A great deal of effort has been made to ensure that this hierarchy does not behave as a straitjacket that prevents the developer from having the full potential that HTML and JavaScript programming offer. Absolutely any **tag** or event of any HTML element can be used in xaWeb without the biggest problem. There are no limitations. Any event, any control can be overloaded in xaWeb. However, xaWeb's main objective is to offer Web solutions to classic business management processes as pragmatically as possible and by simplifying the programmer's knowledge of HTML, CSS and JavaScript programming as much as possible.

The second goal was the automatic, bidirectional data transfer between the web pages created with xaWeb and the CGI application without the developer having to do anything to make this communication mechanism work.

The third necessary objective is that xaWeb could make Web applications in both Linux and Windows environments. An objective that has also been achieved.

Finally, we established a series of requirements that we understood are necessary and involve certain changes with respect to the development of desktop applications for the Windows environment, which are:

- Use of UTF-8 as the only character set: This can be a problem for data environments where data is to be accessed from Xailer desktop applications and xaWeb applications. However, the solution is very simple, at least in MySQL or MariaDB databases since you only must put the "*set names latin1*" statement once you start the connection in your desktop application (internally you will have to create the tables with UTF-8).
- Avoid having to link to any Xailer library. In other words, xaWeb is completely independent of Xailer and does not require any of its libraries.
- The applications had to be 64-bit in Linux since it is the usual environment in that operating system.
- xaWeb should be independent of any Windows APIs, as these APIs do not exist on Linux.

We recommend using the Xailer IDE for application development for several reasons:

- Ease of setting project type: CGI Windows or CGI Linux
- Highly advanced IntelliSense support of all xaWeb libraries
- Debugging in Windows environment

However, those who are used to using the hbmk2.exe tool and a good text editor like Visual Studio Code will have no problem continuing to use their favorite editor.

ARCHITECTURE

xaWeb consists of the following components:

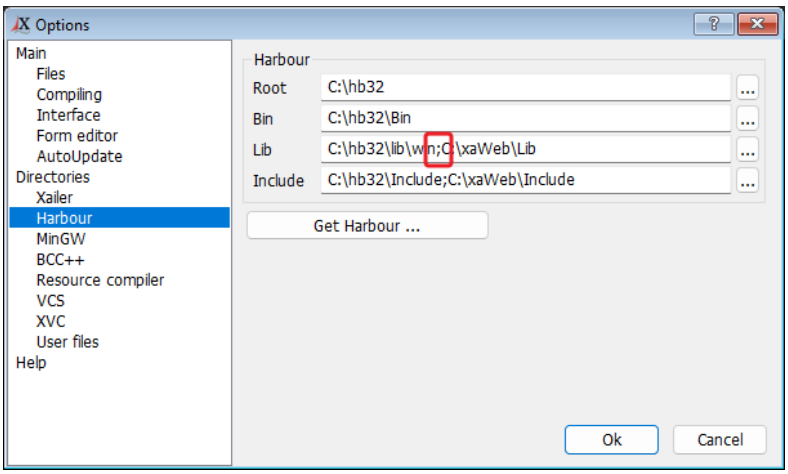
- A static library with all xaWeb source code
- A static additional library to allow overload of classes by the user
- An include file named "xaweb.ch" that you must include in all your modules
- Various lightweight CSS and JavaScript modules

All classes have an empty ancestor that allows the user to overload the class without having to inherit in a new class. An example is easier to understand: The WEdit class represents an Input-Text control of HTML. However, if you go to the xaWeb sources you will notice that the class has only this code:

```
CLASS WEdit FROM ZEdit
ENDCLASS
```

The class that really has all the code is **ZEdit**, but the user never instantiates '**Z**' classes, but '**W**' classes. With this we get that, if the user wants, for example, to add one more properties to the class, he only has to include in his own code a complete definition of the '**W**' class with all its improvements and changes he wants.

We recommend installing xaWeb in the folder 'c:*xaWeb'* as all the examples have that directory indicated for the xaWeb libraries. However, you can install xaWeb in the folder of your choice if you correctly configure the path where the Xailer IDE has to look for the xaWeb libraries. Hanging from that folder you will find the 'lib', 'include' and 'source' folders. In case you use the Xailer IDE, so that your xaWeb projects can easily find the include "xaWeb.ch" file, it is recommended that you include this folder in the Harbour search directories. It would be done as follows:



Notice that both the LIB field and the INCLUDE field have entered a ';' to add more than one search path. Xailer version 9.2 (or Xailer 9.1 Beta 3) is required for this functionality to be available.

When you do projects for Windows, you must indicate in the project the xaWeb libraries for Windows, which are: xaWebWin and wxaWebWin. When you are doing projects for Linux, you should use the xaWebLinux and wxaWebLinux libraries.

To use ***Materialize*** with xaWeb you must include your library named "xaWebMaterializeLinux" or "xaWebMaterializeWin" depending on whether you are making executables for Linux or Windows environment. In addition, you must include the "**xa-materialize.ch**" file instead of "xaWeb.ch" in all PRG files.

IMPORTANT NOTE

The Materialize library must be incorporated into the linking libraries in the project configuration before the xaWeb libraries.

CGI TECHNOLOGY

The xaWeb **development platform** is based on the use of CGI technology for more information I refer to the following links:

- Common Input Interface - Wikipedia, the free encyclopedia
- Common Gateway Interface (CGI): What it is and how it works (godaddy.com)

The main basic functionality of any CGI application is to return the content of an HTML page via standard output (STDOUT). The execution of CGI is done in the same way that a static web page would be accessed through a URL, for example: http://www.example.com/index.html

For the web server to run the application and return the content generated in STDOUT, it needs to know that it is a file that must execute and return the content of STDOUT instead of reading and returning the contents of the file. To do this, you need to use an extension that the web server can recognize as an executable file. In the case of web servers under Windows, you can use the extension 'exe' as usual. On Linux we recommend using the 'cgi' extension.

Applications made by xaWeb, in addition to behaving like a classic CGI that returns the HTML *stream* that the browser will display, are also capable of behaving like a web service that normally returns a JSON object. This type of operation is explained in a later chapter.

The location of CGI executable files on the server should be in ad-hoc folders for that content in which there are only execution permissions on the CGIs that have been installed. In Apache web server the contents of the CGI are in the ***/usr/lib/cgi-bin*** directory.

CGI AS A CODE GENERATOR

It's very important that **you never forget** that your xaWeb code only generates HTML code, JavaScript code, and CSS rules. What runs on your HTML page is the code generated by the CGI, but this is no longer present. It is a very common mistake to try to introduce Harbour code into properties of objects that are then going to be deployed. An example makes it more evident:

```
oButton:OnClick := {|| Msginfo("On click")}
```

This doesn't work. The *OnClick* property of the button must have a code that the browser is able to understand.

PARAMETER SWITCHING TO CGI

The transfer of parameters to the CGI application is usually done through a GET method, which consists of including this information in the URL itself: after the URL itself, the character '?' is included, and then the duplicates of the parameters separated by the character '&' and in the format 'name'='value'. Since URLs do not support special characters and spaces, a simple process of converting the string must be carried out. That process is called 'URL encode' and is done automatically by xaWeb. The conversion of the type of the 'value' element to character type is also performed automatically by xaWeb. Another fundamental method for passing information from the website to the server is through the POST method, which is the one used, for example, when you press the '*submit*' button on a form. In this case, the information is contained in a stream apart from the URL and is received directly by the CGI.

CGI EXECUTION

The CGI initialization code would basically consist of instantiating an object of our inherited ***WDoc*** class that includes all the HTML code that we want to return, period, but obviously this is not possible to do directly, it is necessary to perform a series of operations so that our application can communicate with the web server. The class that is responsible for starting the whole process is the ***WApplication*** class and this is automatically instantiated through an INIT PROCEDURE. This procedure uses an object (singleton) of the ***WEngine class*** that is the only one that communicates with the web server.

In the code of our 'MAIN' function we will indicate the document we want to process. In xaWeb, that document is considered the **default document**, but through the parameter step it is possible to change it to any other document. This would be the classic code of our PROCEDURE Main:

```
PROCEDURE Main()
   Application:cTitle := "Hello World"
   WRouter():New( Application ):Start( "WDocMain" )
   Application:Run()
```

```
RETURN
```

In this example, the class that is going to be instantiated an object is ***WDocMain*** which must necessarily inherit from the ***WDoc*** class. To get our CGI to be able to instantiate a different document, or even execute a specific method of one of the documents, the ***WRouter class*** is used, whose task is basically that: to decide what operation should be performed, with which document, which oversees instantiating and if it must execute a specific method of that document.

When the URL does not include any parameters, the default document is instantiated and its CreateDoc method is executed. This method is responsible for instantiating all the elements that the web page will display.

The Engine Class

Some of the concepts discussed in this chapter require a basic understanding of how the HTTP protocol works.

The ***WApplication object*** instantiates a singleton of the ***WEngine class*** that is stored in the public variable Engine. This object contains important information about communication with the web server:

- Header sent from the web page that is saved in the *hResHeaders property*
- Classic environment variables of the CGI model instantiated by the web server with request information that is stored in the *hEnvironment property.* Such as: AUTH_TYPE, ANNOTATION_SERVER, CONTENT_TYPE, CONTENT_LENGTH, DOCUMENT_ROOT.
- The cookies received are also stored in the *hReqCookies hash*
- Parameters received via GET are stored in the *hParams hash*
- *cPostBuffer* stores the *stream* received via POST. There is also an *hPost* hash that has information for which the stream case has form information. In which case the hash has the values of the form fields.

The **Engine** object holds more useful information that will be explained in a later chapter.

The *WDoc* class is responsible for sending all the information to *WEngine*, which is done automatically when you leave the CGI application. xaWeb automatically controls the entire flow of information to STDOUT. The programmer should NEVER send a piece of data directly to STDOUT, not even through the **Engine object**.

Sessions and Cookies

In web programming, it is necessary to be able to recognize from the web server the user who makes more than one URL request in a short space of time. Cookies offer us this functionality in a very simple way. Basically, a cookie is a file that is created on the client machine but is also accessible from the

web server. xaWeb uses cookies to synchronize the information on the website and the CGI application, as we will see below.

A special cookie in xaWeb is the session cookie, which is a simple ID that maintains its value unchanged until it expires because the CGI URL has not been called for a while. This expiration time is set in the *WEngine:nSessionTTL property.* From the CGI we can access its value through the *WEngine:SessionId method.*

The default internal session maintenance manager is retrieved internally by calling the ***Session***() which returns a "singleton" of the manager, which by default, is an instance of the **WSession**. This object is accessible via the *Document:oSession*. This manager that xaWeb incorporates will be enough for many users, but you can make your own manager and create the singleton function **Session**() that an instance of his personal class returns. This class is responsible for recording and retrieving a *hash* which supplies you with the kind **WDoc** in a JSON file within the ' folder*sessions'*. Analyzing the *WSession*, it will be very easy for you to overload this class so that the methods **Save** and **Load** for example, work with a database instead of flat files.

If you use Linux servers to host your pages, you need to create the '**sessions'** folder from the root '**usr/lib/cgi-bin**' and grant group write privileges to the user 'www-data' (Apache). To do this, you must use the Linux **tools chown** and **chmod** as follows:

```
sudo chown :www-data /usr/lib/cgi-bin/sessions
sudo chmod g+rw /usr/lib/cgi-bin/sessions
```

The base folder where the sessions are located is in HB_DirBase() + "sessions". From this folder, an additional folder is created with the same name as the session ID and inside that folder is the JSON file named "session.json" which is the one that stores all the persistence of the session.

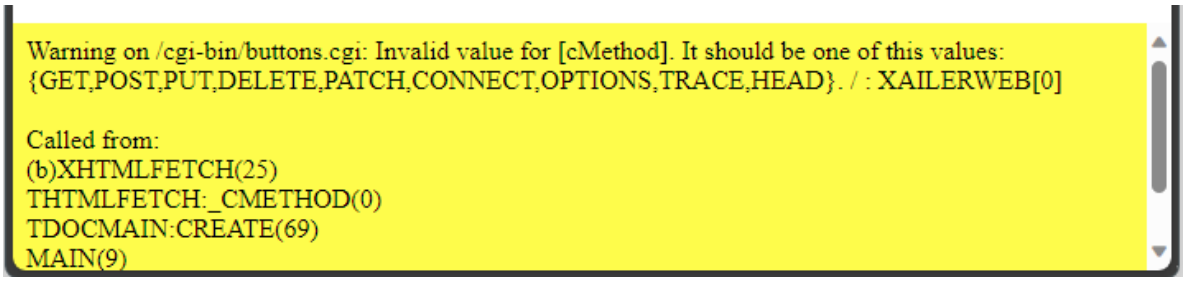
The xaWeb session manager allows you to save everything you want in the session folder. As you have been able to read, a folder is created for each session and, therefore, from your CGI, you can record any additional file you need in that folder. Of course, everything you save in that folder is lost when the session expires. The active session folder is easily accessible with the *Document:oSession:SessionPath()* property.

DEBUGGING MODE

The *Engine* object incorporates a property named *lDebug* that when set to true provides us with the following functionalities:

- It generates a file with all the information that has been sent to the web server, including headers and cookies. The file defaults to 'error.log' in the same directory as the CGI, but can be modified via the *WEngine:cLogFile property*.
- Indicates the CGI execution time in the error.log file.
- Displays warnings of improper use of properties and methods of the various HTML controls.

In addition to this debugging mode, xaWeb incorporates a function called *LogDebug (xVal)* that directly displays in the browser itself (iFrame) any content we want.



It is possible to debug step by step with the Xailer IDE only by making executables for Windows, and this is the only case where you should link to the Xailer.lib library. In all other cases and for the final executable, it is not necessary to link to that library and it is even recommended not to do so.

IMPORTANT NOTE

When you link to the Xailer.lib library in order to debug step by step, it is essential that you prevent the Xailer.lib library from linking in "Check first" mode, which is its default state and can be checked by the name of the library in bold. To switch to normal mode use the context menu of the 'Xailer.lib' element itself. The Xailer.lib library should be linked after the xaWeb and xaMaterialize libraries.

DIRECTORY STRUCTURE ON THE WEB

It is important to be clear about how Web servers serve the pages and where they expect to find the different files that we intend to load. The easiest way to know where the browser intends to find any file is to use the CGI environment variable '*DOCUMENT_ROOT*', which can be easily accessed with the instruction:

```
Engine:DocumentRoot()
```

This is the directory where the Web server (Apache) expects to find all its files. But this directory is not the same one that our CGI will rely on to find other files. As a basic rule:

- Use Engine:DocumentRoot() when dealing with files that need to be accessed by the web server, such as Html, Jpg, Png, Css, Js files. Which is usually: '/var/www/html' (Linux)
- Use HB_DirBase() when dealing with files that you have to access **internally** from the CGI

When we want our web pages to access a resource on the web server, we will only have to put the 'path' from Engine:DocumentRoot(). For example: "/css/style.css" (it would be in /var/www/html/css/style.css). However, if we want to create or save a file in that folder from our CGI, we will have to indicate the complete path.

When we want our CGI to access external files for reading or writing, we must always use HB_DirBase() to indicate the way. If we do not use HB_DirBase() to indicate the path, we will be indicating a relative 'path' from the location of the CGI itself.

It is important to emphasize the importance of whether to put the slash ("/") at the beginning of any relative path. If you put the slash it indicates the root directory of Engine:DocumentRoot(). That is, where the HTML files are located (for example). If you don't put the slash, you are indicating a 'path' relative to the location of the CGI.

On Linux Web servers, CGI is usually set to **`'/usr/lib/cgi-bin'`**, while web pages are set to **`'/var/www/html'`**. Therefore, *Engine:DocumentRoot* points to the latter directory.

However, any file that attempts to load **DIRECTLY** from your own CGI will be relative to the CGI location and not to *Engine:DocumentRoot.* If it does not include any path, the file should be in the same directory as the CGI. If you create a subfolder in the '*cgi-bin'* directory and want to access a file in that folder, we recommend that you use the Harbour function `HB_DirBase() + '/folder'.`

It is not the same for CGI to directly access a file, as it is for the web pages generated by CGI to want to access a file. The difference is total, and it is important that you understand the difference and even more so with Linux servers. Hence the importance that has been intended to be indicated by bolding the word 'directly'.

This problem does not occur in traditional HTML programming (without CGIs), since the initial loading file of the web page that is usually called *'index.html'* is in `'/var/www/html'` and therefore the relative '*paths'* coincide with the absolute *'paths'*.

IMPORTANT NOTE

It is not a good idea for our CGI to generate files that will then be opened by the Web server, such as style sheets, JavaScript code or images, since it is very possible that another instance of that same CGI will modify those files that it has just created. And if they are always the same files, it makes more sense to include them on the website as an external reference.

RUNNING CGIS ON LINUX

In Linux environments it will be necessary to set the necessary permissions for the CGIs to run correctly. Remember that the location of these **is never** where the Html, css, js and images files are located. On Apache servers, under standard installations, the location is in the */usr/lib/cgi-bin* folder, but if you have a Linux server with panel management software, such as *Plesk* or *Cpanel*, it is very likely that you will have to place the CGI files in a different folder and even have to perform some configuration on Apache Web server. You should check your dashboard documentation to know exactly where you should upload CGI files for them to run smoothly. Even if you upload the files to the appropriate folder and correctly indicate the search path in the browser, it probably won't work for you until you correctly set the file's privileges, which should include the '**run' option**. The first step is to set the user 'www-data' as a **group** with permissions in the /usr/lib/cgi-bin folder with the following command:

```
sudo chown :www-data /usr/lib/cgi-bin
```

The *sudo* command asks for admin permissions, and the *chown* command is the one that actually sets the permissions. The next step is to set read and execute permissions on that folder:

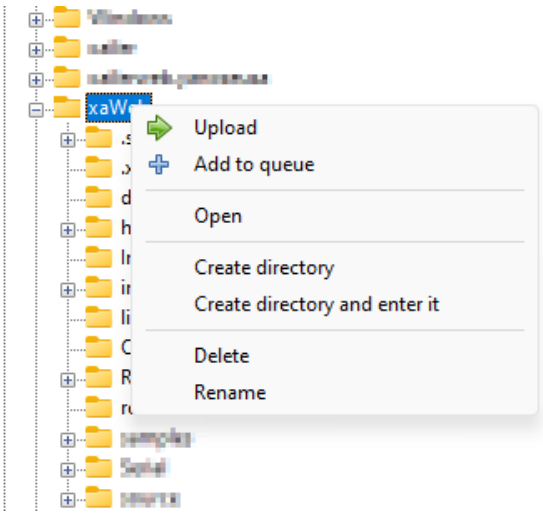
```
sudo chmod g+rx /usr/lib/cgi-bin
```

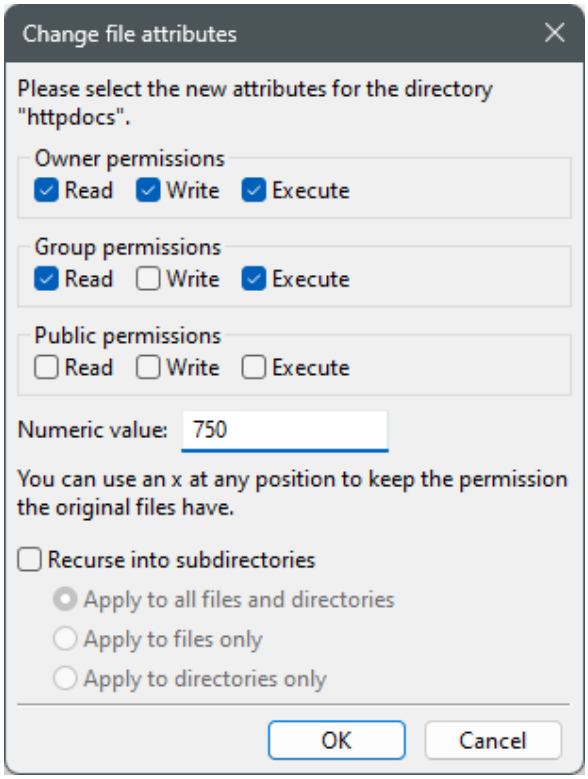
And then you'll need to set permissions on each CGI file:

```
sudo chmod 755 /usr/lib/cgi-bin/myproject.cgi
```

If you receive a 500 error when you try to run CGI from the browser, it is most likely a lack of permissions error. If the browser is not able to find your CGI, you will receive a 404 error.

If you use a tool like Filezilla to upload the files, all this setup is completely simplified using the 'File Attributes' context menu option that the software has:





You will need to set permissions (755) on each new CGI file you upload to your Linux server, as new files you upload will not default to those values. However, this operation will only have to be performed once. The next CGI uploads will maintain the permissions that the file it replaces had. To solve this problem there is a utility called 'setfacl' that allows you to set the default permissions for new files. We recommend the following link for more information.

If you still can't run your CGIs, the next step would be to examine the web server's 'suexec.log' file that in Ubuntu is located at /var/log/apache2/suexec.log. You will need to SSH access your server with the Putty utility or similar.

Don't be impressed by the complexity of all the above. It's easier than it sounds.

The Events

You may already know what an event is and are used to using them. However, events as such do not exist in *Harbour* and you must resort to the use of *code-blocks* to achieve similar functionality. An event is an operation performed by the end user that can be controlled by our application. A classic example is the pressing of a button.

In xaWeb, you must distinguish between two types of events:

1. The ones that are fired from our CGI and are processed by the CGI

2. Those that are fired from the browser, but are processed by our CGI or in the browser itself

The first type of event is the classics already existing in Xailer that many of you will know. This is the case of events:

- oControl:OnPreProcess()
- oControl:OnDeploy(@cHtml)
- oDoc:OnDeployHead(@cHaead)
- oDoc:OnDeployBody(@cBody)

These events can be named after a method in the main document (just like in Xailer) or a block of code. In both cases, the first parameter is always the control or document that triggers the event. The successive parameters will depend on the type of event.

```
Document:OnDeployBody := "MiMetodo"
Document:OnDeployBody := {|oDoc, cHtml| ... }
```

The second type of event is also like the one existing in Xailer, but with greater functionality and important differences:

1. Running a class method: If we specify a literal (oBtn:OnClick := "myMethod") xaWeb will search for a method with that name in the active document. If it finds it, it will cause the CGI to reload when pressing the button, indicating in its parameters that it must execute the "myMethod" method. The method receives an ***hEvent*** parameter that is a *hash* with the ID of the control that triggered the event, the X and Y coordinates of the press, and whether the Control and Alt keys were pressed when clicked.

2. Execution of a JavaScript function: If, as in the previous case, we indicate a literal, but xaWeb does not find any class method with that name, it will cause the button press to cause the call to a JavaScript function with that name.

```
TEXT INTO cJs
  function myfunction(e) {alert('Click triggered from a user function');  }
ENDTEXT

WITH OBJECT WButton():New( Document )
  :cText := "This button fires a Javascript user function"
  :OnClick := "myfunction"
  :cId := "button2"
END WITH

Document:AddScript( cJs )
```

3. Javascript code execution: If we enter a literal between <script> and </script> tags, when we press the button from the web page, the code we have entered will be executed. For example: `<script>alert("OnClick")</script>`.

4. Assignment of an object: If we assign an **object** to the event, it must necessarily have a method named ***Html*** that is the one that will be executed when the deployment is carried out and the value returned by the method will be the one assigned in the deployed HTML.

> **IMPORTANT NOTE**
>
> In the first two cases, for events to work correctly, HTML controls need to have their cID property assigned.

The OnValidate Event

This event is a special event that occurs whenever an <Input> control changes value and when the form submitted is performed. This event will be discussed in more detail in the forms section, but it is important that you take it into consideration from the beginning. Basically, the difference with other standard events is that they must return a JSON object with information on whether to validate the value entered in the control and if not, indicate an error message.

The OnFormShow Event

This event is a special event that occurs whenever a form is displayed. This event will be discussed in more detail in the forms section, but it is important that you take it into consideration from the beginning. Basically, the difference with other standard events is that they must return a JSON object with the initialization values of each data entry control.

The Doc and DocSection class

As mentioned in a previous section, the *WDoc* class is responsible for displaying HTML content. An xaWeb application can have multiple *WDoc* objects, but only one of them can run each time the CGI is executed. Through the parameter pass, we can specify exactly the *WDoc* object that we want to instantiate and, therefore, it will be responsible for displaying the HTML content. Once a *WDoc* object has been instantiated, it is accessible via the public variable '**Document**'.

The *WDoc* object can be responsible for the display of all HTML content, but it is convenient to use it to separate each piece of content. For example: the header could be one section, the footer another, and the body, another one. Sections in xaWeb are created using the *Document:AddSection(<cName>)* method and return an object of type *WDocSection.* Even if you haven't created a section, xaWeb automatically creates a default initial section with the name "*default".*

A *WDoc* object can have multiple sections or *WDocSection* objects. Even two different WDocs can share multiple sections, as would be the case with sharing a footer. Just because you create a section doesn't mean it has to be deployed. The *WDocSection* object has a property named ***lDeploy*** that when true indicates that the section should be deployed.

All sections are displayed in HTML inside an HTML container of the type <x-doc-section></x-doc-section>. The name indicated in the AddSection() parameter is used as the ID in that HTML element. Another important property of the WDocSection object is ***lHide***. This property allows you to hide the section, but still display it: simply style '*display:none*' to the HTML element. The usefulness of this property is to be able to create hidden sections that we can make visible by JavaScript code at will, such as a form.

The order of creation of the sections marks in principle the order of display of these in HTML. However, it can be changed by using the ***lFooter*** property that forces the section to follow the main HTML body.

When CGI is executed without any parameters, only the *default WDoc* object is instantiated, and a call is made to its *CreateDoc* method. After the call to *CreateDoc*, the document is deployed and exited from the application. However, through GET parameters, we can indicate a different document or a specific method to execute. For example:

> http://localhost/test.cgi?**action=mydoc-mymethod**

This URL prompts the loading of the '*mydoc*' document and the execution of its '*mymethod*' method after it has been instantiated and its CreateDoc method has been executed.

When we need one or more sections to be instantiated after from the call to '*mymethod*' and Not before, we can use the ***RegisterSection(<cName>)*** of the class *WDoc* (*lDeploy* a if false), which receives as its only parameter the name given to the section. The advantage of using this method is that the recorded sections are instantiated after of the method chosen in 'action' and therefore they can access already updated data.

SYNCHRONIZATION OF WEB AND CGI ENVIRONMENTS

xaWeb incorporates deterministic JavaScript code (which is always the same) into all its executables, which can be embedded in the application or in a separate JavaScript module (xw_backpack.js). This module contains a few functions that xaWeb will use internally without the programmer having to know how they work. In addition to this deterministic code, xaWeb includes an additional non-deterministic script that varies based on the xaWeb code that has been executed. This script basically performs the following operations:

- Create variables to indicate the HTML elements on which you want to persist
- Handle events (e.g., OnClick) of JavaScript-level HTML elements that have been overloaded in the CGI

To achieve the persistence of certain HTML elements and therefore that some of their properties are visible from the CGI, the *Harbour clause* 'PERSISTENT' is used when defining the property in its class.

With an example it is seen more clearly:

```
CLASS ZEdit FROM WInput
  DATA cValue INIT "" PERSISTENT
ENDCLASS
```

With the PERSISTENT clause at the HTML control level we are indicating that we want the '*value*' property of that HTML element to be accessible in the CGI the next time it is executed with the updated value of the web page. The next time you run the CGI, you will see how the *cValue* property of that HTML element has automatically recovered the value it had on the web page.

The xaWeb classes that support HTML elements come pre-designed with the values PERSISTENT in the class members that in principle are considered to have persistence. If you want to add persistence on any other member, you must overload the Z class by indicating the PERSISTENT clause on the additional members you want. For example:

```
CLASS WEdit FROM WInput
  DATA lEnabled PERSISTENT
ENDCLASS
```

xaWeb includes an additional mechanism to make any property persistent at **the application level** and that is by using the PERSISTENT clause at the WDoc level. For example:

```
CLASS WDocMain FROM WDoc
  DATA cUserName INIT “” PERSISTENT
```

Internally, all the synchronization of values is done through cookies that are generated automatically. In the '*Engine*' *object* several hashes are created for this purpose, which are:

- *hState* that stores all the values of HTML elements of type PERSISTENT.
- hEvent that stores the properties of the event that has been triggered from the web page. This *hash* receives the ID of the control that triggered the event, the X and Y coordinates of the press, if the Control and Alt keys were pressed when clicked. You may receive additional information about the context in which it is being used.
- hCargo that stores additional values that may be required in some type of event. For example, it is used in editing operations of records from an HTML table, where hCargo contains the values of the columns in the table, which can be modified and returned to be updated in the HTML table.
- And those already mentioned above from hParams and hPost that give information about the parameters passed and the fields of a form if it had been sent.

The ***Create*** method that all xaWeb HTML controls have is responsible for assigning the PERSISTENT values with the *hState* hash information of the *Engine* object and therefore has to be executed after its properties have been initialized:

```
With Object WEdit():New( oSection )
  :cValue := “” // opcional  . . .
  :Create()
End with
```

IMPORTANT NOTE

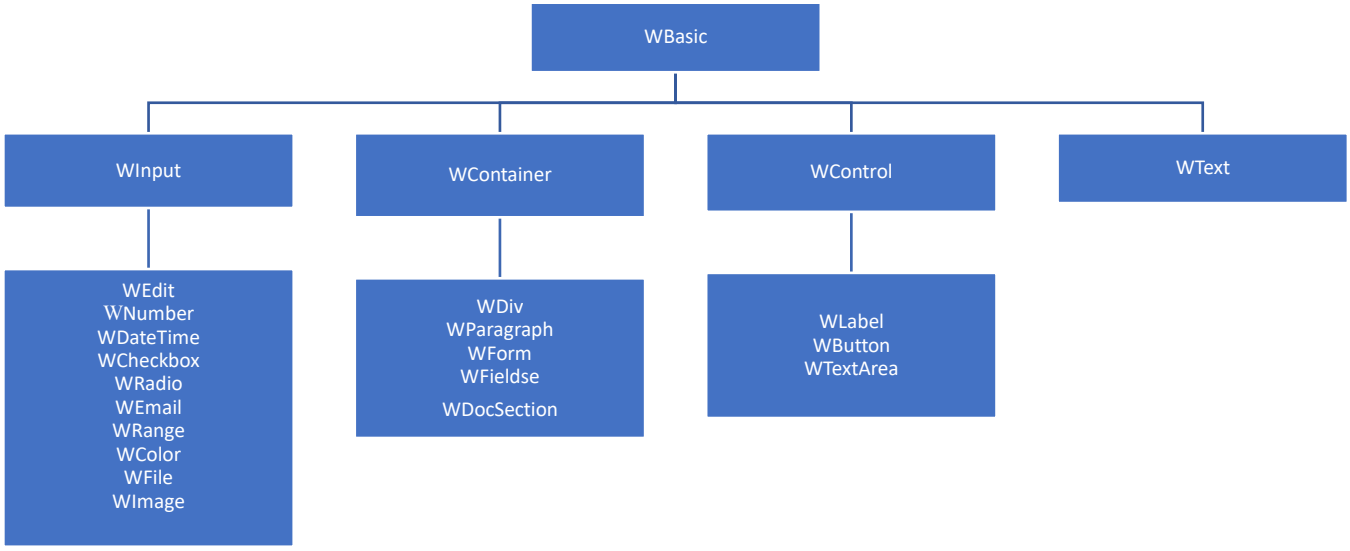
For HTML control persistence to work properly, the controls must be assigned their cID property and their Create() method executed.

CLASS HIERARCHY

So far, these are the classes we've discussed about xaWeb:

- Class *WApplication* with its corresponding public variable *Application*.
- *Class WEngine* with its corresponding public variable *Engine* that is responsible for communications with the web server.
- WRouter *class* that is responsible for performing the initial routing of the application, establishing the type of operation to be processed, the WDoc object to be instantiated and the method to be executed of it.
- WDoc *class* with its corresponding public variable *Document* that is responsible for doing all the HTML deployment.
- WDocSection *class*, which are objects that represent the different sections that a *WDoc object can store*.

In xaWeb, every HTML element is a Harbour object. Even to put a simple 'Hello World' it is necessary to create a special xaWeb object and this is because the user MUST NOT send textual content directly to STDOUT as PHP does with the ECHO command. The WDoc object that is instantiated is solely responsible for displaying the HTML content.



This is the basic class hierarchy of any xaWeb HTML control. An editing control such as *WEdit* inherits from *WInput and WInput* inherits from *WBasic*. The *WContainer* class encompasses all controls that can have more HTML controls.

The *WDoc* object contains an array of *WDocSection* objects, and each *WDocSection* object contains an array of aControls objects*, which if they are of the* WContainer type can have more controls. When HTML deployment is requested in WDoc through its ***Render*** method, it causes HTML deployment in all WDocSection sections, and each section in turn does the same with its arrays to Controls, which may even have more nested controls.

This entire deployment process is done by calling a *RunHtml*() method that all controls have and at all inheritance levels. For example: The WEdit *RunHtml* method will do what it needs to do, then it will call the *WInput* RunHtml *method, and finally the* WBasic *class's* RunHtml *method will be called*.

There can be a **1-1** relationship between an HTML element and an xaWeb control. For example, the <h1> element corresponds to a single *WText*() control in xaWeb. But this is not always the case. For example, the *WTable* control has multiple internal child controls that are themselves HTML elements.

This relationship can get even more complicated when the control that represents the HTML element becomes a child control of a new control that has been instantiated in its constructor. With some examples, the concept is clearer:

- By default, in HTML, <table> elements do not have vertical scroll bars. That is, the HTML element will vertically occupy the sum of its header, its rows and its footer. How can we get the control to have a specific vertical length and show us a scroll bar? Simple, we include the <table> element in a with fixed or ruled dimensions via CSS and that this is the one that shows the scrollbar. Therefore, it is necessary that when we create a <table> element, we also create an element which is the one that will contain the element <table>. In the case of the *WTable* control, that element is accessible through its *oContainer property*.

- The *WEdit* control could correspond to a single HTML element <input>, but in xaWeb, it has much more functionality since it incorporates the possible <label> and an error message when the value entered in the control is invalid. When we instantiate a WEdit object we are instantiating a container of type WDiv accessible as oContainer, which in turn contains an accessible WLabel object such as oLabel for the description of the field, an accessible WSpan object such as oError to show the possible error, and finally a text-type WInput object which is referenced in the constructor's return.

- This complexity of multiple automatically instantiated controls is further complicated when using frameworks such as Materialize.

This automatic creation of controls can be a problem, especially when you intend to iterate through all the controls of a *WDoc* object; to avoid this problem all controls have two properties that indicate their inherited ancestor: **oParent** and **oOwner**.

The *oParent* property sets the parent-child dependency, in the same way as HTML. That is, when we create the *WTable object* inheriting from a *WDiv*: The oParent of *WTable* would be *WTable:oContainer* and the parent of *WTable:oContainer* would be the *WDiv*. As you can see, we've inserted an *oContainer* object in between the two. To **iterate** over all the controls, even those created automatically and in the same way as the HTML document, we will use the *aControls* array that *WDoc has* and all the controls that they inherit from *WContainer*.

On the other hand, the *oOwner* property indicates the control that really creates the control. To **iterate** on these types of controls we will use the *aComponents* matrix that *WDoc has* and all the controls that they inherit from *WContainer*.

PRE-PROCESSED (PRE-DEPLOYMENT)

As we have already explained before, all the deployment of HTML code occurs when WDoc:Render is executed automatically and it executes the *RunHtml* method in all its inherited controls, starting with all the sections and these in turn execute it in all its HTML element type controls, which in turn can have more controls.

Before this cascading execution of the *RunHtml* method occurs, the same is done with a method named **PreProcess**. As a result, before the RunHtml method is executed, all controls process its *PreProcess* method, which exists at all inheritance levels in the class hierarchy. Each control executes its PreProcess method, and when it is finished it calls Super:PreProcess to continue its cascading processing. The *WBasic* class is the last in the hierarchy, and only executes the **OnPreProcess** event that allows the user to add all the code they deem necessary.

This *PreProcess* method is very important. In it, most controls make significant modifications to the deployment of the control (and dependent controls). They can even create new controls that will also

be deployed. These controls that are created in this method are scoped on the object only in the *OnPreProcess event*.

TYPES OF CGI OPERATIONS

CGI behaves differently depending on how it has been executed:

- Without any parameter: In this case, the CGI will generate the Web page of the HTML document that it has defined by default.
- Through parameters that are passed by GET-type command. GET commands that are sent in HTTP requests are those that are included as additional text in the URL itself. For example:

  http://localhost/test.cgi?**action=mydoc-mymethod**

  The first parameter passed by the GET command sets the type of operation. In this case it would be "action" and its value "mydoc-mymethod". This example would ask the CGI to instantiate the "mydoc" document and execute its "mymethod" method. As it is an "action" type operation, we know that it has occurred within an event on our website and therefore we will receive as a parameter in the method a *hash* with information about the event: Who triggered it, mouse coordinates, keyboard press status and some additional interesting data. That parameter is actually the *Engine:hEvent property*.

  xaWeb currently receives four different types of operations via GET commands, which are:

  1. "Action" **type operations**, which are those that are produced by events on the web page, such as the press of a button.
  2. "Form" **type operations**, which are those that occur when performing the '**submit**' of a form. In this case, the parameter that is received is a *hash* with all the information entered in the form.
  3. "Service" type operations, which occur when requesting a Web service, either from our application or from third parties.
  4. "Custom" **type operations**, for any other type of operation. You will need to use the Engine*:hParams hash* to know exactly what information has been passed via GET.

All possible operations will be explained in depth in future chapters.

PASSING PARAMETERS FROM CODE

We've already looked at the types of operations that xaWeb CGI currently support, now let's look at how we can set those parameters from our own code. The types of operation that the programmer

can set directly are 'action' and 'service', since the 'form' type is generated when a 'submit' button is pressed on a form. The manual way to set a parameter would be, for example:

```
oLink:hRef := “https://www.example.com/mi.cgi?action=mydoc-mymethod”
```

That would generate the HTML so that, when you click on the link, the CGI will be loaded with the parameters we have indicated. xaWeb has two methods to set the parameters easily, which are:

- oWDoc:Action(<cMethod>, [<cDoc>])
- oWDoc:Service(<cMethod>, [<cDoc>])

```
oLink:cHRef := ::Action( “myMethod” )
```

While it might seem that these two methods return a string, they actually return objects of type **WTask**, which offer more functionality than a simple URL string, such as allowing additional parameters to be added to the URL with the *AddParam(cName, xValue method).*

The *WLink:chRef* property is a method of type **SETGET** that always returns a string, but can receive a string or an object. The **WTask** object is stored in the *WLink:oHRef property*.

Remember that in the case of events you only need to indicate the name of the method:

```
oBtn:OnClick := "myMethod"
```

JAVASCRIPT

In xaWeb there is a specific class for the management of scripts named **WScript**(), which can be instantiated either by indicating a URL or the code directly. By default, the script will be placed in the footer of the page, but with the ***lFooter*** property to false you can force the script to go in the header. Another important property is ***lDefer*** which allows you to indicate that the script should be loaded once the page has been fully loaded. This property is really only contemplated by the '**script' tag** in the *header*, however, in xaWeb it also makes sense to use it in scripts that go in the *footer* and this is due to the need to be able to control the scripts that have to go before or after the *Javascript code* that xaWeb adds on all pages. Examples:

```
Document:AddScript( “http ….”, cName ) → WScript object
```

WScript objects have a property named '**cName**' that allows each of the modules to be identified. This property allows, if the programmer wishes, that all the script code is retrieved from an external file instead of being incorporated into the CGI. When a script is assigned this *cName* property, xaWeb will search the server for the existence of a script with that name in the '**js**' folder. If it finds it, it will include a reference to that file, rather than including the entire script code.

It is also possible to create JavaScript code directly from any module of your application and deploy it in the resulting HTML. For example:

```
TEXT INTO cJs
  function myfunction(e) {
   alert('Click triggered from a user function');
  }
ENDTEXT

Document:AddScript( cJs )
```

CSS

With the same functionality as indicated for Javascript, xaWeb incorporates a class expressly for the management of CSS called WCss that allows you to include references to external CSS files or directly introduce CSS code.

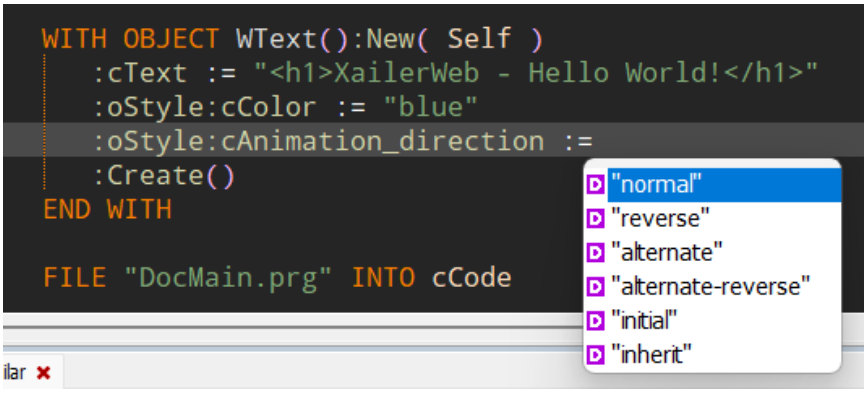
```
Document:AddCSS( cText, cName, lUri ) → WCss object
```

WCss objects have a '**cName**' property that allows each of the modules to be identified. This property allows, if the programmer wishes, all 'CSS' code to be retrieved from an external file instead of being incorporated into the CGI. When a *WCss* object is assigned this *cName* property, xaWeb will search the server for the existence of a CSS file with that name in the '**css**' folder. If it finds it, it will include a reference to that file, instead of including all CSS. You can add more CSS code with the oCss:AddCode(cText) method.

xaWeb aims to follow best programming practices and this includes extensive use of CSS and encourages its use. However, it is always possible to set a style directly on top of the control with the AddStyle(cText) method that all controls have.

```
With Object WText():New( Document )
  :AddStyle( “color:red” )
  :cText := “Hello”
End with
```

Inline styles can also be entered through the oStyle property that all controls have. As soon as the user references that property, an object of the **WStyle** class is automatically instantiated, which incorporates almost all the available styles, and in cases that are listed, it even shows a list of the possible values:



Using inline styles with oStyle is fully compatible with the AddStyle() method and can be used together, without issue.

In addition to this oStyle property, there is another property accessible with the name oContext, which is an instance of what is called a '*Context helper*', i.e. a helper to the context package you are using, which will be discussed in a later chapter. Currently, the only *'Context package'* that includes a '*Context helper*' is **Materialize.** Its use is very similar to oStyle and it is supported by Xailer's Intellisense without any problem. The oContext object is a simple wizard so that you don't have to remember the names of classes and styles that each context package imposes. By way of example (Materialize):

```
oDiv:oContext:Col( 12, 4, 2 )
```

It would be equivalent to establishing the class "col s12 m4 l2"

A later chapter explains in more detail the system that xaWeb uses to integrate frameworks such as Materialize.

Web Services

Web services allow you to establish communications with other applications and retrieve any type of data. The request for the service occurs asynchronously and without leaving the website. Therefore, it is the website that must process the request when it has been concluded. For more information, please consult the following link.

xaWeb allows you **to consume** any third-party web service and even behave itself as a web service. Note that the option to consume refers to consumption on the website itself and not to consumption from within the CGI. In addition, we must distinguish between **consumption of third-party web services** and **consumption of services of our own CGI** or other CGI created with xaWeb, since the latter type offers greater functionality.

To consume any third-party web service, you just have to create a **WFetch object** indicating the URL of the web address you want to access and assign it to an event of a control. When an external third-party web service is requested from your website, you will typically receive a JSON object, which it can process. Please note that processing is necessarily done in JavaScript and therefore a minimum knowledge of JavaScript is required to run third-party web services.

This small example runs a *Web service* of the web *ip-api.com* that allows us to know the location of our public IP:

```
oFetch := WFetch():New( "http://ip-api.com/json/example.com" )

WITH OBJECT oFetch
```

```
   :cTargetId := "mycity"
   :cSourceId := "button3"
   :cCallBack := "citySolver"
END WITH

WITH OBJECT WButton():New( Self )
   :cText := "This button calls an asyncronous API"
   :OnClick := oFetch
   :cId := "button3"
   :Create()
END WITH
```

The constructor parameter is automatically assigned to the object's *cUrl* member. The members to assign to the *WFetch* object for this type of operation are:

- *cTargetID:* With the ID of the control that will receive the information
- *cSourceID:* With the ID of the control that fires the event. This is used to leave the control disabled while the Web service that is asynchronous is running. Optional.
- *cCallBack*: JavaScript function that will receive the JSON object returned by the web service.

```
TEXT INTO cJs
   function citySolver(element, data) {
      if (data.city) {
          element.innerHTML = 'Example server is at ' + data.city;
      } else {
        element.innerHTML = 'City could not be found';
      }
   }
ENDTEXT

Document:AddScript( cJs )
```

It is very likely that the web service will require some additional parameter via **GET**, which consists of including after the URL, the character '**?**' followed by the parameters in the form: 'key=value' and separated each of them by the character '**&**'. The easiest way to include such parameters in the URL is to use a **WTask** object, which is explained in a later section.

In the previous paragraph we explained how to include parameters within the web service call; but if you analyze it carefully, you will realize that, in many cases, it is useless, since the parameters of the web service will depend on values that the user enters on the web page, and, therefore, values that the CGI does not know. However, xaWeb offers a solution to this problem: With xaWeb, it is possible to pass parameters of existing values on the website. You'll need to use the **WFetch**:AddJsParam(cId, cAttribute) method. 'cId' and 'cAttribute' correspond to the **ID** and **attribute** of the HTML control you want to pass, which will usually be '**value**'. For example: **WFech**:AddJsParam("idEdit", "value"). If a non-standard attribute is specified, xaWeb will try to find that attribute as the '**dataset**' of the control itself. For more information see this link.

The use of the *WFetch control* that we have seen so far is mainly characterized by the fact that it returns a *data stream*, which is usually of the JSON type, although it could also be XML or directly a

binary *stream*. These types of FETCH operations are of type ***cContentType*** = *application/json; charset=utf-8*.

If we want our CGI to behave like a web service or web service provider we will have to pass the " command "**service**" indicating the class name of the document **WDoc** and the method you want to run separated by a hyphen. After that first GET parameter, you can include as many as you want and even use the POST method additionally if you wish. For example:

http://midominio.com/miapp.cgi?service=mydoc-mymethod

IMPORTANT NOTE

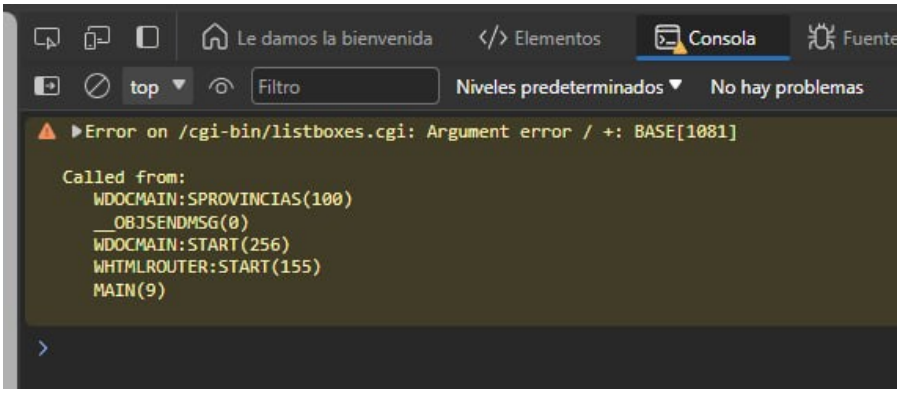
xaWeb can function as a web service, but it should be noted that every time you run an xaWeb service, you run our CGI, you offer the service, and you automatically **exit** the application. Any modifications you have made to your program are lost. You have to think that your program is not really running, but rather, a separate service that your program offers, but that nothing or almost nothing has to do with it.

The only parameter that the method will receive is a *hash* with all the parameters sent. The method must return a stream. Normally it will be a JSON string (cContentType = application/json), but it can also be directly JavaScript code that will be interpreted by our website as we will see later.

xaWeb incorporates a powerful mechanism to avoid having to process the value returned by the web service from the web page, but this mechanism only works with web services offered by CGIs made with xaWeb and consists of directly returning JavaScript code that will be processed directly by the web page when the request is received. Just set the cContentType property of the *WFetch object* to "***application/javascript***" and logically you no longer need to specify the *cCallBack property*.

Any modifications you make to any HTML controls property within your CGI will be internally converted into JavaScript instructions that will be executed on the web page.

When the error occurs within a 'service' (type "service") there is no web page on which the error information can be displayed. In such cases, the errors will only be displayed in the browser console with messages such as 'console.warn(...)'.



THE WTASK CLASS

This class allows you to manage in an agile and simple way the URLs that your application is going to require. Depending on the context in which it is used, one constructor or another must be used:

- To indicate an operation type "*action*" to a method of any document, we would use the constructor WTask:Action(<cMethod>, [<cDocument>]), where cMethod is the name of the method to be executed and cDocument is the document to be loaded, which by default will be the one set by default in the application.
- To specify a "*service*" operation: WTask:Service(<cMethod>, [<cDocument>])
- To specify an operation of type "*other*": WTask:Other(cOperation, cDocument)
- To directly indicate a URL we will use: WTask:Url(<cUrl>)

The **received WTask** object must be assigned directly, in the same way that you would assign ownership of the URL. For example:

```
oLink:cHref := oTask,
oButton:OnClick := WTask:Action("myMethod")
```

The **WTask** class includes the AddParam(<ckey>, <cValue>), which allows you to easily add parameters of type GET to the URL, and the SetParam(<nPos>, <cValue>) method to change the value of any parameter.

It is possible to map directly to a control type **WLink**, for example, the text resulting from the URL containing the **WTask** object using the WTask:Html() method.

PACKAGES

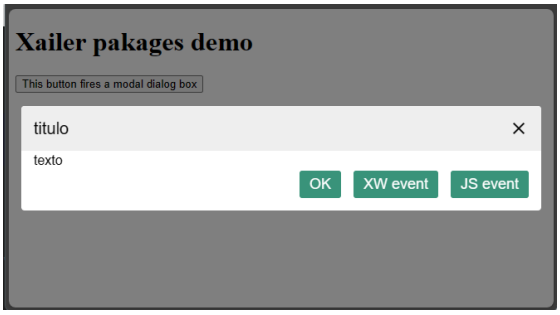
xaWeb's package management is a super powerful tool that allows you to easily extend the functionality of your web pages. Packages are nothing more than a CSS and JavaScript code wrapper

that has the extra functionality of creating ad-hoc objects for said package that can relate to xaWeb. It is seen more clearly with an example:

```
oPackage := WModal():New( Self )
oModal := oPackage:ShowModal( "Items", "Confirm deletion", { "Yes", "No" } )
oModal:OnClick( 1, "DeleteItem" )

WITH OBJECT WButton():New( Self )
  :cText := "Confirm deletion"
  :OnClick := oModal
  :cId := "button1"
  :Create()
END WITH
```

This example shows a modal dialog that if the 'Yes' button is pressed executes the document's 'DeleteArticle' method. For more information see example 10-Packages.



All packages created for xaWeb must inherit from the ***WPackage*** class that includes the minimum functionality that a package must have, which basically consists of adding CSS and JavaScript, either through URLs or directly by code.

The classes that inherit from ***WPackage*** must allow instantiation of objects that depend on it and must incorporate an **Html** method. That's all. Adding any free or commercial web component via packages is a very simple task.

CONTEXT PACKAGES

Context **packages** are specially designed packages when you decide to use xaWeb with an additional ***framework***. These packages are responsible for configuring all the colors and general CSS that the application will use. The packages are designed to be easily overloaded by the programmer and can be adapted to your own taste. By simply incorporating the package into our document, its appearance will be completely modified.

Currently xaWeb includes four packages:

1. **WaterContext**: Based on Water.CSS that is designed to make websites simpler, more beautiful, responsive and with theme control. We recommend using it when premium showiness isn't important.

```
WITH OBJECT WWaterContext():New( Self )
   :cTheme := “auto”
END WITH
```

2. **SimpleContext**: Based on Simple.CSS that has the same purpose as the previous one, but with a different style. This style has automatic theme support, but it doesn't allow you to easily change it by code.

3. **BasicContext**: This package is designed so that the user himself is the one who designs the main characteristics of the colors that his application will use, both in light and dark mode and establishes the default CSS values of each of the Tags that his application will use. It is completely customizable.

4. **MaterializeContext**: This package includes the complete Materialize framework, which not only includes theme management and responsiveness, but also includes a lot of proprietary components and controls that make creating web pages much easier.

All context packages included in xaWeb are configurable by the programmer through CSS variables and its *WContext:cCssMods* property that allows overloading the initial CSS values of the package. Each context package uses its own variables that initialize to the values that they deem appropriate, but that can be changed without problems.

Unfortunately, each package uses its own variables, and the breakdown of available colors varies greatly between packages. At xaWeb, we wanted to unify the colors that a web application can use. Regardless of whether each package uses its own variables, with depth in the development of these, we have established the basic colors that a business management web application should have, which would be the following:

1. Background color, alternate background, text, and muted text on the page (body)
2. Background color, text, and primary hover
3. Background color, text, and secondary hover
4. Background color, text and accent
5. Edge color
6. Gradient Color

The alternate page background color would be used for panels, cards, and disabled effects; the primary color for links, buttons, focus effect, and active form states; the secondary color for headers and footers, the highlight color for controls and areas that you want to highlight, the border color is used for lines and borders and finally the gradient color is used to set the opacity/transparency effect.

Regardless of the context package used, all these colors are accessible by the programmer through methods of the WContext class:

```
METHOD BodyColor()            INLINE "var(--body-color)"
METHOD BodyAltColor()         INLINE "var(--body-alt-color)"
```

```
METHOD BodyTextColor()         INLINE "var(--body-text-color)"
METHOD BodyTextMutedColor()    INLINE "var(--body-text-mutted-color)"
METHOD PrimaryColor()          INLINE "var(--primary-color)"
METHOD PrimaryHoverColor()     INLINE "var(--primary-hover-color)"
METHOD PrimaryTextColor()      INLINE "var(--primary-text-color)"
METHOD SecondaryColor()        INLINE "var(--secondary-color)"
METHOD SecondaryHoverColor()   INLINE "var(--secondary-hover-color)"
METHOD SecondaryTextColor()    INLINE "var(--secondary-text-color)"
METHOD AccentColor()           INLINE "var(--accent-color)"
METHOD AccentHoverColor()      INLINE "var(--accent-hover-color)"
METHOD AccentTextColor()       INLINE "var(--accent-text-color)"
METHOD BorderColor()           INLINE "var(--border-color)"
METHOD GrColor()               INLINE "var(--gr-color)"
```

To change any color in **WBasicContext**(), you have four options:

1. Manually modify the CSS file: *xw_BasicContext.css*
2. Set the new colors in the **WContext**:aDarkColors and **WContext**:aLightColors arrays

```
DATA aDarkColors  INIT {"#00001a","DarkSlateBlue","white","LightGray",;
                        "Navy","cornflowerblue","white",;
                        "MediumVioletRed","HotPink","DimGray",;
                        "Gold","PaleGoldenrod","Black",;
                        "LightSteelBlue", "64"}

DATA aLightColors INIT {"white","whitesmoke","black","gray",;
                        "Tan","Wheat","black",;
                        "Teal","LightSeaGreen","DimGray",;
                        "SkyBlue","LightBlue","Black",;
                        "Bisque", "255"}
```

3. Use the cCssMods property to overload the values you want. For example: `--body-color: white.`
4. Change the color using the **SetColor**(cName, cValue, lDark) method

All these changes are at the **context** level of the entire application. If you need to change the color of a simple control remember to just do: `oButton:AddStyle( "color: White;" )`

In your CSS code you will use for example: var(--body-color), while at the CGI level you can use oContext:BodyColor(). These colors only exist in their entirety for the W**BasicContext**() context package. It is very likely that other context packs do not support some of these colors and their use is useless. We recommend checking the code of each context pack, and the colors that are supported.

The basic context package (WContext), from which the rest of the context packages inherit, is responsible for setting the display theme: *'none', 'light', 'dark' or 'auto'* with your property cTheme. As a '*theme*' has been set, the package stores that information with the name 'theme' in 'Session storage', in a 'Cookie' and in a global attribute of the document (document.documentElement). Such information is accessible from any context package.

It is also possible to create countless user colors from any context package using the WContenst:AddCustomColor(cName, xLightColor, xDarkColor) method, which can then be assigned to any control:

```
oControl:oStyle:Color := ::oContext:CustomColor( “my-color” )
```

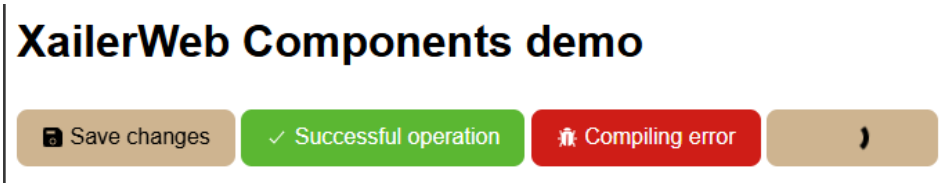
PDF PACKAGE

xaWeb comes with a dedicated package for printing any type of document in PDF format that is easy to use. This package uses a template system to easily create any type of report. Currently (Oct 2024) the automatic printing of any HTML table is supported. The 'invoices' template that this bookstore also offers will soon be incorporated.

WEB COMPONENTS

Web components are blocks of code that encapsulate the internal structure of HTML elements, including CSS and JavaScript, thus allowing the code to be reused as desired in other websites and applications. xaWeb allows the use of any third-party web component, facilitates the creation of web components and simplifies their use as much as possible. You can see an example in "samples\Comp-buttons". Here you can see the **WCmpButtonIcon, WCmpButtonSpinner, and WCmpNumericKeypad controls**.

Web components have the great advantage that they encapsulate all the necessary code (including JavaScript and CSS), **on the contrary**, it complicates access to the possible internal elements of the component. Therefore, its use should only be done in very clear cases where a component is the most desired. The example we give is a good starting point for the xaWeb community to start making their own components and make them accessible to the community or sell them online.



THE TABLE CONTROL

We have paid special attention to the HTML table component, including in it and optionally everything that an xBase programmer is used to finding in this type of control, but without neglecting the possibility of setting any special settings that are needed in the tables. We recommend that you have a minimum of knowledge of the HTML Table control before continuing your reading.

The basic HTML system of defining all the rows in the table is unusual in xBase and impractical. It is preferable to load the information with the LoadData() method that includes the ***WTable*** class. This

method receives an array with all the rows in the control. Each row is an array of columns. This method can be called multiple times. In fact, it is normal to call it first to set the header of the columns (*header*), a second time to set the data (*body*) and a last time to set the footers of columns if they exist (*footer*). The *nHeader* and *nFooter* properties accept a numeric value that indicates the number of rows in each of these sections.

The **WTable** control uses internal objects to save all its information, so that it is possible to set new styles or properties to each of the sections that an HTML Table can have, such as:

- *aHeaders*: Array of *sTableCol* objects that represent the table header
- *aRows*: Array of *sTableRow* objects that represent each of the rows in the table
- *aColGroup*: Array of *sTableColGroup* objects required to set the ColGroup HTML properties
- *oHeader*: An sTableZone object that defines the table's header zone
- *oFooter*: An sTableZone object that defines the footer zone of the table
- *oBody*: An sTableZone object that defines the data zone of the table

Each cell in the table can be accessed. For example:

```
oTable:aRows[2]:aCols[1]:nRowSpan := 2
```

The events OnStartRow(oSender, oRow, nType) y OnStartCol(oSender, oCol) allow you to access each of the rows and cells before your HTML code is generated so that you can set any style or property on each of them. These events are only triggered when you feed the table from the CGI with an operation such as '**action**'. Operations type '**Fetch**' that directly return a JSON to our website it does not make much sense to trigger such events since it is not possible to perform any operations on the DOM on the page.

By default, in HTML, no maximum length is set on tables. All its rows are displayed consecutively, and it is on the website itself where the scrollbars are established and not within the 'browse'. To ensure that the length of the board is at maximum height, it must be placed inside a container (div) that has a limited height. There are other ways to do it, but in my opinion, this is the most recommended. For this reason, the WTable element always has a container, which can be accessed with its **WTable**:oContainer property. Therefore, the oParent of a **WTable** object is always its oContainer object.

To perform the classic addition, edit and deletion operations, it is necessary to have a reference to each of the **WTable** rows. xaWeb uses a very simple technique that consists of using the first column of the table as the identifier of each of the rows. The LoadData() method is given a second parameter that allows you to indicate whether the first column is a row identifier or not. The lShowID property (default to false) allows you to indicate whether or not you want that column to be visible.

Several properties have been added to the WTable object to simplify its use, which are:

- *cHeaderBkColor*: Header background color
- *cHeaderColor*: Color of header text

- *cFooterBkColor*: Foot Background Color
- *cFooterColor*: Footer Text Color
- *lResponsive*: To make the table responsive and display as cards on mobile devices
- *lCanSort*: To allow the table to sort by any column
- *lCanFilter*: For the table to allow filtering by any column
- *lShowSelected*: What the selected row is displayed for

Two properties that deserve a separate mention are *oDelControl* and *oEditControl*. These two properties allow you to indicate the controls that may exist for deleting and editing records. The buttons will remain disabled until you have selected a row from the table. The *lShowSelected* property must have a true value.

FORMS

xaWeb has made it as simple as possible to use forms. You only need to specify the method to use and set its **cName** property to match the name of a method in your ***WDoc*** object. The 'cAction' property does not need to be set, unless you want to call another URL that has nothing to do with xaWeb.

xaWeb has chosen not to have a specific INPUT type-button control as it is the same as the HTML element <button>, but with less functionality as it does not allow HTML tags to be set in its text. For this reason, it is important that when you include a button in your form, it has its cType property at the value "submit".

Buttons in HTML have a 'type' name property that allows you to set the type of the button. These are:

- button
- submit
- reset

xaWeb incorporates another type of '***cancel***' name used to automatically set the action to be performed when clicked. In this way, the programmer does not have to assign his *OnClick* event. Finally, the '*cancel*' type buttons are generated as a '*button' type*.

The event that triggers form acceptance is the form's ***OnSubmit*** event. When a control type **WButton**, its type is indicated as '*submit*'; when clicked, in addition to its own '*OnClick', the form's* OnSubmit event is fired, if it is overloaded.

By default, this type of form causes a 'FORM' type operation which involves calling a new URL from the browser and therefore the loss of all the content of the current page. xaWeb also incorporates the option of being able to submit forms via FETCH processes and therefore does not leave the current page. This type of form is explained in a later chapter.

xaWeb has a control for each type of HTML **Input element**. For example, **WEdit** corresponds to a 'text' type input (text, password, tel, url and search). There is a **WEmail** control that corresponds to an "email" type input, and so on to all input type controls. In HTML programming it is necessary to create an additional "label" control to assign input to each element, and, in fact, if this control is not created, the browser usually indicates a warning for its absence. xaWeb automatically creates such label controls automatically. This does not mean that he does not have access to it. In fact, you can set any style, class, or whatever you want to that **Wlabel** control. All controls of type 'input' have the ***oLabel property*** (with the class 'xw-input__label'). In the same way that an automatic label is contemplated, xaWeb also includes a div to indicate a possible name validation error *or Error* (with the class 'xw-input__error') and that of course is also accessible to modify its style. When you create an input in xaWeb, you create the input and three more controls, which are:

1. A **Div** that encompasses all controls, including the input and whose parent is the oParent indicated in the creation of the input control (class 'xw-input')
2. The 'label' element whose **oParent** will be the '**div**' created in point 1
3. The input element itself, whose **oParent** will be the div created in point 1 (class 'xw-input__input')
4. The '**div**' element to show the possible error, whose **oParent** is the '**div**' created in point 1

The classes set in the controls allow you to set their look easily through CSS.

A special case, and as in Xailer, is the **WRadMenu** control, which is a single control that encompasses several 'radio-buttons' that work together. With a simple control, the entire structure indicated above is created for each of the 'radio' type elements of the control. The control allows vertical or horizontal alignment and is very easy to use.

Types of Form Operation

This type of operation occurs when the '**Submit**' button on a form is pressed. The '**cName**' property of the form indicates the name of the method to be executed in the CGI. This type of operation causes a reload of the web page including the information in the form:

`http://example.com/test.cgi?form=wdocmain-myform`

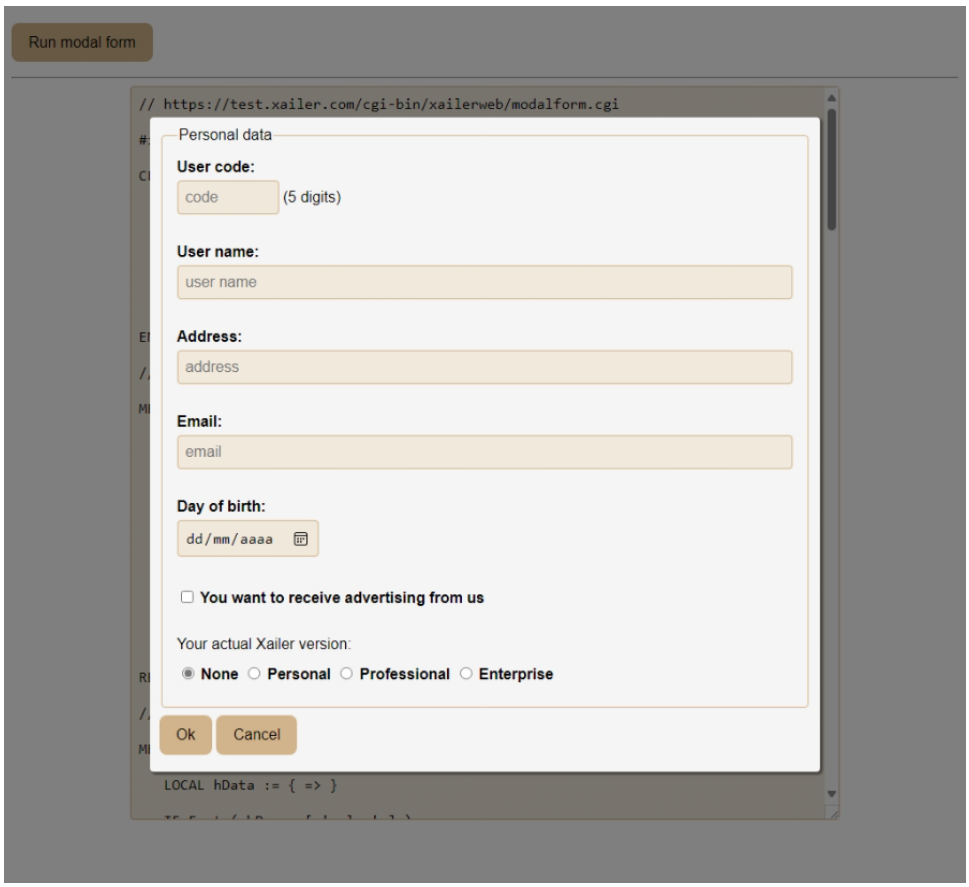
The method receives as its only parameter a hash (hPost) with all the information of the form fields.

Advanced use of Forms

The forms we've seen so far have two simplicity features:

- They are not displayed above the existing web page
- Invoking, accepting (pressing OK) or cancelling (pressing Cancel) the form requires a reload of the original page indicating a certain action (Action)

The **WForm** class has a **lModal** property data that allows it to display the form floating and centered above the web page.



This property gives a great visual aspect to the form, but since it is centered on the screen (viewport) it is likely to be cropped on mobile devices if the form is very long and then vertical scroll bars are displayed.

To display the form above the page you can do it in two ways:

1. Run a process '**action**' to load the form Adding A new **wDocSection** when an event occurs (example 13-ModalForm)
2. Load the form always, but hidden, and only show it when necessary (example 14-ModalFormFetch). In this case, the form is loaded along with the main section into a new **WDocSection**, setting its lDeploy and lHide property to true. The button that displays the form must call in its OnClick event the **ShowSection**(cName) method that all the controls have because it is defined at the level of the **WBasic class**.

With the first option, the state of the current website can be lost as a new URL is loaded by the browser. However, with the second method you don't get lost. The Onclick event action of the 'Cancel' button that you have incorporated into the form with the type 'cancel' (ctype='cancel') will be automatically adjusted depending on the way you have chosen to display the form.

All 'input' **type controls** have an additional event named '**OnValidate**' that allows you to validate the existing text in each of them. This event is executed automatically when the control changes its value, but also when you try to 'submit' it. The event, as in all xaWeb cases, can be processed at the JavaScript level or at the CGI level:

- If it is resolved at the JavaScript level, you will receive two parameters in the function you have specified: the **value** of the control and the **HTML element**. You will need to return a simple JSON with the property 'pass' with true or false and another property 'error' (optional) with the description of the error produced.
- If it is resolved at the CGI (Harbour) level by means of an operation such as '**service**', will receive in its only parameter **hParam** the values: **'Value'** and '**Id'**. You will need to return a simple JSON with the property 'pass' with true or false and other property 'error' (optional) with the description of the error produced. The easiest way to do this is by creating a Hash with the keys 'pass' y 'error' and then return the JSON with the **HB_JsonEncode(** hHash **)**. You may also receive another value in the hash with the name "append_mode", which can be true or false. This only happens when a table is connected to a form. An option that is discussed below.

In the example 14-ModalFormFetch, in addition to using the hidden loading system of the form, includes an extra functionality, which is to carry out the entire process through an operation such as 'service', which means that the website is never really abandoned. To achieve this functionality, you just have to overload the event OnSubmit of the form using the **SubmitToService(** cMethod **)** of the form itself:

```
oForm:OnSubmit := oForm:SubmitToService( "MyFormData" )
```

The **SubmitToService** method (cMethod, lJson) indicates the method of our CGI that should be executed when the form has been accepted and whether it should return a JSON object to be processed by xaWeb after editing. By default lJson is false, which is the most common case when **you simply** want to update the HTML content of a web page element, as is the case in the example 14-ModalFormFetch. When **a JSON object** does not return, the Fetch operation is done of the type "application/javascript" and therefore our service will automatically take care of generating the JavaScript code so that the elements that have been modified in our CGI are also modified in the web page.

As it is a service, the method receives a parameter of type hParam. That is, a Hash with the parameters sent in the **Fetch** operation. However, as it is a form, it is normal for that information to be sent through POST, so the parameter received is useless. This is not a problem, as all POST information can be obtained from the **Engine:hPost** property.

The form class has an additional event named '**OnFormShow**', which is basically used to be able to set initial values in each of the Input controls that the form has. This event, when assigned with a literal matching the name of a method in the document, triggers a 'Service' process (Fetch operation)

executing that method. The **Engine:hCargo** property contains a Hash object with all the IDs of the input controls and their current value. You will only have to modify the ones you want and return the hash again using the hb_JsonEncode(hHash) function. You can see an example of using this event in Samples\16-Form-init.

USING MASKS IN INPUTS

An important feature that is missing when we abandon the Harbour language is the use of masks with the 'Picture' clause that standard GET-type controls have or in the case of Xailer, the cPicture property that many of the editing controls have.

A library has been incorporated into xaWeb to allow masks to be available in the <type controls**input.text**> with functionality very similar to that which exists in Harbour. The selected library has been Maska, but as with sessions, the user is allowed to select any other library for this purpose. The default internal mask manager is retrieved internally by calling the **InputMask**() which returns a "singleton" of the manager, which by default, is an instance of the **WInputMask**.

WEdit-type controls incorporate a property named '**cPicture'** that allows you to set the control mask. Here are the different template elements that the property accepts:

1. Maska's own elements:
    a. '#': Digits 0 to 9
    b. '@': Letters not including international characters
    c. '*': Digits and letters not including international characters
2. CA-Clipper-style xaWeb elements (an element corresponds to an entered character):
    a. 'A': Letters including international characters
    b. 'N': Digits and letters including international characters
    c. 'D': Digits
    d. 'U': Letters including international characters in capital letters
3. xaWeb elements other than CA-Clipper (one element corresponds to one or more characters entered)
    a. '0': Digits
    b. '9': Optional digits
    c. 'B': Digits and letters including international characters
    d. 'V': Digits and letters including capitalized international characters

    Examples:

    - "0.99": Numeric of any length and two decimal places
    - "#99.#99.#99.#99": IP address
    - "B B": Two words
    - "V V V": Three words in capital letters

4. xaWeb function elements, for numeric values only:

    a. "!#ll:dd:p" Where:
        - 'll' is the local identifier of the format. Default to the one indicated in InputMask():cLocale
        - 'dd' is a numerical value with the number of decimal places to be used
        - 'p' is a numerical value that if other than zero, will only admit unsigned values

THE CACHE

xaWeb is very fast, but it can be even faster if caching techniques are used, which in the case of xaWeb, can be done at the control level. For example: a 'select' type control that encompasses all countries in the world. Each time you load the website, you must access a database, load the countries and integrate them into the control. We know that countries are not going to change for a long time, therefore, it makes a lot of sense to specifically search for that control. xaWeb is capable of caching not only individual controls, but also controls that encompass other controls, such as the footer of the web page. xaWeb supports two types of cache:

1. Global: which applies to any request
2. Session-level: Applies only to the current session. That is, each session has its own cached controls.

And its implementation is very simple, you just have to indicate the value '*global*' or '*session*' to the property oControl:cCache. That's all.

In both cases, the cache is distinguished by language, that is, there is a cache version for each language that is being used and has been designated in the Document:cLang property.

If you want to override the cache completely, it is recommended to use the Document:FlushCache() method only once. If you want to clear the cache of a certain object, it is preferable to set the cCache property to the value '*flush*'.

IMPORTANT NOTE

You should only cache controls whose content does not change anything between calls. Even the simplest variation prevents its use.

TABLES AND FORMS: COMPLETE MANAGEMENT

A significant effort has been made to make CRUD-like operations that rely on HTML tables easy to use. The complexity involved in communication between CGI and the website, without losing the information on the website, means that all operations must be carried out without reloading the page and all the processes of registration, modifications and deletions are carried out through *Web services* provided by CGI itself.

The joint management of tables and forms to make a complete CRUD entails some complexity when it is intended to be done completely with **Fetch services**. That is, without having to reload the website. These would be the necessary steps:

1. Create the form as indicated in Example 14-ModalFormFetch. That is, loading the form section in stealth mode and using the WForm:lModal property to true.
2. Display the form assigned to the OnClick event from the button that triggers it by the WButton:ShowSection(cSection) method.
3. Indicate in the WForm **OnSubmtit** event the method of our **WDoc** that we want to be triggered for both registrations and editing: **WForm:SubmitToService**("Srv_Method", .T.) passing as an additional parameter a logical value **.T.** which indicates that we're going to **return** a JSON object in the method.
4. Set the cTableId property of **WForm** to the ID of the table you want to process
5. Set the lShowSelected property of **WTable** to true so that a row can be selected
6. **Optional**: If you have created buttons for 'Edit' and 'Delete' you can set the oEditControl and oDelcontrol properties with the identifiers of both buttons. This will automatically disable them when a row at the table is not selected.
7. Set the cDataField property of all WInput **type controls**. The value must match the value you gave to the fields in the table.
8. If there are any fields that you do not want to be editable in edit mode, set the lDisabledOnEdit property of that field to false.
9. Set the WTable:Append(cFormSection) method to the **OnClick** event of the button that performs the registration. It receives as its only parameter the name of the section where the form is located.
10. Set the Edit(cFormSection) method of WTable in the OnClick event of the button that performs the edit. It receives as its only parameter the name of the section where the form is located
11. To delete a record, you do not need to show any form, you just need to call the service that deletes it. In the 15-Tables CRUD example we have chosen to display a confirmation message relying on the **WModalMsgBtn** package.

The method assigned in the WForm **OnSubmit** event will be responsible for validating the registration and editing operation and will allow you to do all the internal operations on your database. In this method you receive all the information in **Engine**:hPost. Specifically:

- Pairs of values from all fields in the form. Its name matches the cDataField property that you entered in point 4.
- Original value pairs for all fields in the record. Identical name to the previous one, but with a prefix 'old_'.
- Key "append_mode" with true value if it is a registration operation.

The service that deals with the operation of deleting records must return a simple hash with the 'pass' key to true or false, as desired.

The 25-Tables CRUD example shows you a complete CRUD of a table. If you look closely, you will see that there is no persistence in the database located on the server. It is done like this, he best, so that the original data is not lost due to the tests that users can do.

MULTI-LANGUAGE MANAGEMENT

xaWeb incorporates a package called **WTranslator** that is responsible for this task. For a control to be translated into another language, it only needs its *lTranslate* property to be set to true. As soon as this occurs, a single instance of the **WTranslator** class is created that will be responsible for translating all the controls that have the *lTranslate* property to true.

The **WTranslator** class is supported by a DBF table located at *HB_DirBase()* + "***data***" with the same name as the CGI, and with the following configuration:

- File structure: {{ENGLISH},"C", 255, 0}, {SPANISH},"C", 255, 0}}
- Index structure: TAG(1) Name: ENGLISH, Expression: PADR(ENGLISH, 255)

The internal translation manager is retrieved internally by calling the **Translator**() which returns a "singleton" of the manager, which by default, is an instance of the **WTranslator**. This translation manager that xaWeb incorporates will be enough for many users, but you can create your own manager if you wish.

We have opted for a DBF table to save translations because we understand that it is the fastest and most convenient system for xBase users.

If you use Linux servers to host your pages, you need to create the '**data**' folder from the root '**usr/lib/cgi-bin**' and grant group write privileges to the user 'www-data' (Apache). To do this, you must use the Linux **tools chown** and **chmod** as follows:

```
sudo chown :www-data /usr/lib/cgi-bin/data
sudo chmod g+rw/usr/lib/cgi-bin/data
```

SYNTAX HIGHLIGHT

xaWeb comes with a package called **WSyntaxHilite** that makes it easy to highlight **source** code. It is supported by the HighlightJs JavaScript library and for its use it is only necessary to instantiate a WSyntaxHilite object in the main CGI document: `WSyntaxHilite():New(oDoc)`.

To use it, simply insert the source code preceded by the <pre> and <code> tags:

```
<pre><code>Hilite this</code></pre>
```

xaWeb installs a special version of the CSS of this library to allow it to work correctly with the theme manager (light-dark) of the context manager. The structure of the WSyntaxHilite class allows it to be adapted to your needs by changing the colors it uses.

For more information on all the possibilities of this library, access the following link.

FILE <cFILE> INTO <cVAR>

In the examples, you can see that we use this command quite often. This is the mechanism we can use to replace the resources we use in desktop applications. It is possible to add any type of file, even binaries, but if you want to include it within a control you must use the **HB_Base64Encode()** function for its conversion. And at the JavaScript level, you'll need to use the **xw_b64toUnicode() function.**

THE ECHO AND OS COMMAND

The **ECHO** 'text' command that xaWeb includes simply converts the command to the following code:

```
WText():New(:__WithObject()):cText := 'texto'
```

Which creates a **WText** object and assigns it its cText property. That's all. Notice the **WText** constructor receives a ':WithObject()' value, which refers to the object that is inside a WITH OBJECT block. For this reason, if you use the **ECHO** command outside of a WITH OBJECT block, you will receive a compilation error. In this case you can directly indicate the parent of the constructor with the command: **ECHO** 'text' **INTO** oContainer.

The '**OS**' command is the same as typing:__WithObject() and is intended to be the acronym for STACK OBJECT.

WSL

We encourage (vehemently ;-) to use WSL (Linux Subsystem for Windows) with Ubuntu for the creation of executables in Linux environments and their subsequent execution from an Apache browser (also running on WSL). The new version 9 of Xailer is prepared so that the entire process of

creating and booting the browser with CGI is carried out automatically. More information in our wiki in https://en.wikipedia.org/wiki/Windows_Subsystem_for_Linux.

Remember that it is necessary to indicate in the output file the complete WSL path where the CGI will be created, which is: \usrl\lib\cgi-bin\{output-file}

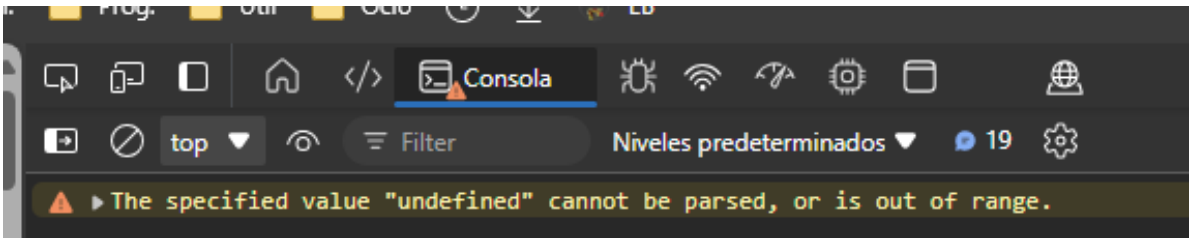
Note: At the time of writing (Nov-24) Xailer's integrated browser does not support FETCH application/javascript operations, nor visualization of the source code of the page and if the development mode is invoked in the browser, background processes are generated that consume the entire CPU. For all these reasons, it is recommended to use WSL, which offers a work environment as friendly as Xailer's integrated browser.

THE JAVASCRIPT AND HTML LANGUAGE

It's not a difficult language for an xBase developer to understand, but admittedly it can be very frustrating at times. The most important thing to know about Harbour is that he is case sensitive in almost everything and therefore it is very important that you pay attention to that aspect. A control ID with a value of "First" is not the same as a value of "first". My recommendation is that you always use lowercase letters for everything that has to do with setting control properties. I assure you that a lot of problems will be avoided.

You don't need JavaScript to use xaWeb, but sooner or later you will lose your fear and start performing small functions to avoid, above all, unnecessary page reloads. I encourage you to look at xaWeb's JavaScript code and try to understand how it works. Through any Internet search engine, you will find a multitude of information on how to do almost anything with JavaScript.

The code you make will no doubt fail at first, but all browsers include very powerful programming tools that shouldn't intimidate any Harbour programmer. Simply press F12 in the browser, go to the console and check for the error. It's easy to put breakpoints and go step-by-step to find out where the error occurred. Lots of encouragement.



It is likely that despite not programming a single line in JavaScript, you will have no choice but to go to the debugging tool to check the console if an error has occurred, since you notice that your program does not act as expected. For example, when a field in a form does not collect data from a table record. In most cases, it will be a simple identifier not found issue or something similar.

IMPORTANT NOTE

When you make changes to a JavaScript module, it is almost always necessary to cause a page refresh by pressing Shift+F5.

xaWeb JavaScript functions

xaWeb provides two more communication mechanisms from JavaScript with CGI. The first of them is using the JavaScript function ***xw_setHbData***. This function sends additional information in the next '**Action**' or '**Form**' type request. Remember that the value of all control properties that have the '**persistent**' clause are passed on to the CGI each time the CGI is executed. This feature allows you to add any additional data from any element of the web page. This function receives the following parameters:

- HTML Element (Object)
- Name of the DATA that you want to set the value for. For example: "cValue"
- Value
- If true, the data will be sent by POST method instead of cookies. False default

xaWeb uses the Cookie mechanism for the constant transfer of information between the CGI App and the website. Unfortunately, cookies are limited to only 4 Kbytes per domain. For this reason, the previous function has the option of sending the data through an HTTP POST operation that has practically no limit and is received in the same way in the CGI App. The sending of data has been compatible with this function and the submission of forms by POST method.

The second mechanism is instantaneous execution through a Fetch process, which is reduced to the call of a simple JavaScript function named **xw_GetHbData**. As the name suggests, this feature retrieves any data from your CGI directly, **without leaving the website**. The value returned by the function must be processed from our website and depending on how you have set the recovery system in its second parameter, you will receive a JSON that you will have to process, or you will receive JavaScript code that will auto-execute on your website. Here are the parameters of the function:

- Name of the method in our **WDoc CGI** document.
- Name of the document itself (Wdoc:cName). Optional, by default the current document.
- Operation type: "application/json; charset=utf-8" or "application/javascript". Optional, by default, the first one.

The Examples

The examples have been numbered so that they can be tested and analyzed in the same order as indicated. They start with a simple 'Hello World' and get more complicated little by little, trying to show all the possibilities of xaWeb.

The examples do not pretend to be spectacular in terms of showiness and even the former do not use any type of Framework, precisely to show the essence of the code, both at the PRG and HTML level.

You can see all the examples running on a Linux server at: All together

The License

xaWeb is a commercial product, owned by OZ Software. However, the free demo version that includes 90% of the programming sources is fully operational and it is feasible to carry out small projects with it. When the project gets bigger, you need to acquire the business license.

The price of the commercial version has not yet been assigned, but in any case, it will be a very tight price that we expect to be around 150 euros and there will be a launch offer with a significant discount.

The license is perpetual and comes with an annual subscription that includes:

- xaWeb updates for twelve months from purchase
- Technical support on our forums
- Access to an 8-hour introductory xaWeb online course

Subscription renewal costs 50% of the commercial license value at the time of renewal.

It is possible that, as with Xailer, in the future, there will be different versions of xaWeb, depending on the functionality they offer.

The Deployment

The deployment consists of copying to the folders located in the cloud, where the web pages, executables and any other file necessary for our web application to work correctly will be hosted. This work can be done manually via any FTP client. However, the Xailer IDE has a specific **plugin** for this made by me and you can get all the information about it at this link.

Collaboration

A small group of alpha/beta testers is created that receive periodic improvements and bug fixes made in xaWeb. During this phase, all but a couple of modules of xaWeb sources will be delivered for intellectual property safeguarding purposes.

We are interested in knowing, above all, the classic processes that users require in their Web management software, to try to simplify them as much as possible, discussing in the channel the best way to do it.

If you would like **to collaborate** with xaWeb, we encourage you to join the alpha/beta testers group by sending an email to the mailto:iozuniga@ozs.es address requesting it. We emphasize the word 'collaborate', because it will be necessary for their collaboration to be active. Otherwise, we reserve the right to remove you from the list of beta-testers.

IMPORTANT NOTE

If you are going to use the Xailer IDE to create your xaWeb CGIs you will need Xailer 9.1.

THANKS

I would like to thank:

- Domenic Corso
- Kevin Powell

The HTML, CSS, and JavaScript videos of these authors have been very inspiring. Significant portions of its code have been incorporated into xaWeb with appropriate copyright acknowledgements.