developer manual

version 0.1.0

Index

1. Introduction

This document is an introduction to the application development for the eyeOS web Operating System. Some PHP knowledge is required to read this document, and it is not necessary to have experience programming with CSS, JavaScript or XHTML, although knowing these technologies will help you understand better eyeOS and its internal functioning.

1.1 What is eyeOS

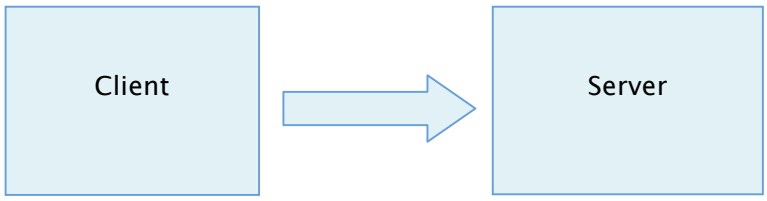
From a technical point of view, **eyeOS** (http://www.eyeos.org) is a platform for web applications, created with the idea to make easy the application development.

There are currently a lot of web-related technologies, such as PHP, XHTML, CSS and JavaScript, so it is required to master a lot of languages and understanding numerous concepts to be able to create web applications. In addition, every web browser has a different interpretation of the code and every PHP version and configuration works slightly different from the others.

eyeOS intends to cover those and other problems derived from the web development, offering the programmers a homogeneous platform to develop their web applications, using only PHP code and leaving to the system the resource management, the communication with the browser, the security, etc.

1.2 eyeOS basic structure

Before studying the eyeOS components, we must know its basic structure. The platform is created over a client-server architecture, in which eyeOS is the server, and the client is usually a web browser.

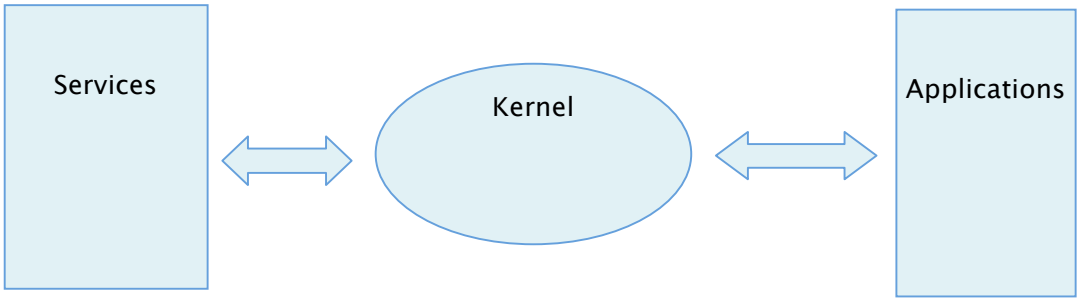


Note: Unless specifying the contrary, this manual will assume the client is a web browser.

1.3 The Kernel

The first step we must take to know eyeOS is the study of its kernel.

Since eyeOS is a microkernel-based platform, the kernel is only thought to unify the system services. In other words, eyeOS has many services for specific tasks, and the kernel is responsible for its communication and location.



With this architecture, applications don't need to specify how to invoke a service, they just need to know its name, and the kernel does the rest of the job.

1.4 System Services

The Services in eyeOS perform low-level tasks. For example, the applications do not manipulate files directly: instead of that, they use a service designed for that purpose. Thus, security is guaranteed since the services apply restrictions to the applications' requests.

The only form to communicate with a service is the kernel, as stated above in section **1.2**. To do that, the service() function (defined in system/kernel/kernel.eyecode) must be used.

service() takes 3 arguments:

```
service( string $servicename , string $functionName [, array $params] );
```

The first argument specifies the name of the service to call. In eyeOS there are eight services, and they will be studied later.

The second argument corresponds to the function to call, given that a service is a collection of functions and we must specify what function we want to use. For example, if we are using the filesystem manager service, we need to specify if we want to copy, delete or create a file.

The third and last argument is an array containing the parameters passed to the function. For example, if we are using the filesystem manager service, we need to tell it which file we want to delete.

An example of a call would be:

```
service('vfs','delete',array('file.txt'));
```

This code would call the vfs service (the filesystem manager) to delete a file named file.txt.

1.5 Libraries

The libraries in eyeOS are similar to the services. They too are called using the kernel, but unlike the services, they do not handle low level tasks (such as files, users or processes), instead they make the development of applications easier, making available sets of functions the applications might need.

eyeOS includes lots of libraries (that will be studied later). Libraries can be used by any application. For example:

eyeZip – allows to work with ZIP files (extract, compress...)

eyeURL – allows to request URLs

eyeXML – allows to work with XML files (read, write...)

The way to use a library is very similar to that of a service, being the only difference the fact that the *reqLib()* function is used instead of *service()*. *reqLib()* takes the same arguments as *service()*:

reqLib(string $libName , string $functionName [, array $params]);

The first argument is the name of the library to use, such as eyeZip.

The second argument corresponds to the function to use, such as extractZip (of eyeZip).

The third and last argument are the parameters passed to the called function.

An example showing the use of this function to extract a ZIP file to a directory using the eyeZip library would be:

```
reqLib('eyeZip','extractZip',array('file.zip','Documents/'));
```

1.6 eyeOS code organization

It is important to know how eyeOS is organized to be able to develop applications for. In eyeOS everything has its right place to be put on.

The eyeOS directory tree can be summarized as:

eyeOS/accounts – Information and credentials about the users. See chapter 2.2.

eyeOS/apps/ – Directory where the applications are located. See chapter 4.

eyeOS/extern/ – Files visible from the exterior, such as CSS stylesheets, images, etc. See chapter 2.6.

eyeOS/extras/ – Extra files. See chapter 12.

eyeOS/groups/ – Groups directories. See chapter 6.

eyeOS/i18n/ – I18n and eyeOS translation directory. See chapter 3.3.

eyeOS/log/ – System logs directory. See chapter 2.7.

eyeOS/system/ – System directory. Contains the services, libraries, kernel and global configurations.

eyeOS/tmp/ – Directory where files are stored temporarily.

eyeOS/users/ – Users' directory where all their documents and files are stored. See chapter 2.2.

The code of eyeOS is stored in files with the .eyecode extension (for example: kernel.eyecode)

which contain normal PHP code, usually using the eyeOS Toolkit classes and functions.

1.7 Settings.php

One of the most important files that defines the eyeOS functioning is the settings.php file (located in the eyeOS root, along with the index.php file). In this file constants are defined and later used in all the eyeOS code. For example:

```
define('ROOTUSER','root');
```

Defines the name of the root user (the administrator of the system).

All the settings of this file will be studied later in this manual.

1.8 Versions

eyeOS releases a new version of the platform every 2 months for normal versions, and every 3 months for special versions.

The version numbers always use 4 digits. For example:

```
1.5.0.1
```

Where the first number (1) indicates the major eyeOS version, the second number (5) indicates the minor version, the third number (0) indicates the sub-version and the last number (1) indicates the revision.

Thus, eyeOS 1.5.0.1 indicates that the eyeOS is type 1 serie (the previous eyeOS belonged to the 0 series), version 5 (the fifth version of the eyeOS 1 series), sub-version 0, revision 1, which means that some bugs have been fixed from the 1.5.0.0 version.

When a new eyeOS version is released, it may contain some errors. Those errors are fixed and a new version is released, changing only the last version number and adding no new features, only patches to fix bugs.

An example of this would be the **1.5.0.6** version, a fixed (patched) version of eyeOS series 1, version 5.

2. Basic services

As explained in chapter 1.3 there are 8 services in eyeOS, which are:

- **extern**
- **eyeX**
- **log**
- **mmap**
- **proc**
- **sec**
- **um**
- **vfs**

They are located in the **eyeOS/system/services/** directory.

2.1 VFS, Virtual File System

One of the most important services in eyeOS is **VFS**, which is the responsible of providing functions to work with files and directories.

Absolutely no operation with files must be performed directly using PHP functions such as **fopen()** and **unlink()**. The VFS service must always be used in these operations.

VFS guarantees that a user is unable to read or edit another user's files. If an application executed by a user requests VFS to read a file over which the user does not have enough permissions, VFS will return false and will activate the error code **VFS_INSUFFICIENT_PERMISSIONS** (read chapter 3.1).

Thus, it's VFS that ensures the security and not the applications. The applications do not need to worry whether the user has enough permissions on a file.

The VFS internal work is done in a modular way. This means that the VFS functioning can be modified using new modules. The default module is the *virtual module*, although it can be specified which module to use changing the value of the **VFS_MODULE** parameter on the settings.php file.

The virtual module uses two files for every file the user creates. One file is used to store the content of the file and other file is used to store information about it using the XML format.

In addition, the virtual module concatenates a 32 character length random string to every file name, making it impossible to guess from the outside the name of a file.

For example, if we call VFS with the virtual module to create a file:

```
service('vfs','create',array('file.txt'));
```

The system will actually create two files, being one called:

file.txt_[RANDOM_STRING].eyeFile, which will store the content of the new file.

And another called:

file.txt_[RANDOM_STRING].eyeInfo, which will contain information about the new file, such as its author, its real name (file.txt) and the creation date.

VFS always manages these low-level tasks. For example, if we want to delete the file called file.txt we just need to call VFS once and both files will be deleted:

```
service('vfs','delete',array('file.txt'));
```

If we want to open the file to read it:

```
$fp = service('vfs','open',array('file.txt','r'));
```

and from here we can use the file resource **$fp** using **fread()**, **fwrite()** and **fclose()** from PHP. The opening modes of VFS open are the same as the **fopen()** function used in PHP.

Apart from this method using pairs of files, the virtual module has methods to deal with normal files which do not use the eyeInfo and eyeFile pairs. These methods are called real methods.

The real methods work exactly like the virtual ones, but their name is preceded with the prefix real_. Let's see an example:

```
service('vfs','real_create',array('file.txt'));
```

This code would generate a file called file.txt in the system. It would **not** create an eyeFile and eyeInfo file.

The reason there exist real methods (real_open, real_create, real_delete, etc) and virtual methods (open, create, delete) is that a user's files are stored as virtual files to control their creation date, their author and other parameters, while the system files, such as a user's configurations, are real files.

As stated in chapter 2.2, every user has a home directory to work in. Inside every home directory there are:

- **conf**/ – the configuration files are stored here as real files with XML format.
- **files**/ – the virtual files (using eyeInfo and eyeFile) are stored here, and are visible to the user from the File Manager.

The form to decide whether to use a virtual or a real method is as simple as thinking: “Do I need to work with a user's file or with a configuration file?”

One detail to bear in mind is that the real functions do not have the same restrictions as the virtual ones.

The virtual functions have permissions to read and write the files in the files/ directory of a user's home, and to create files in the groups' directories where the user belongs (see chapter 6). Inside a group's directory, the virtual functions can only edit and delete files that belong to the user.

The real functions have permissions to create and edit files everywhere in a user's home (including

files/, conf/, etc), but they have no read nor write permissions on the groups's folders.

These restrictions exist for security reasons. It would be dangerous that real files of groups could be altered, so they can only be accessed with virtual functions.

2.2 UM, User Manager

The UM service manages the users in the system and provides the methods necessary for registering, logging in, obtaining the path to a user's home, etc.

One important characteristic is that UM provides a global variable accessible at any place, called **$currentUser**, which contains the username of the current user. For example:

```
global $currentUser;
```

This way, applications can know which user executes them.

Furthermore, UM provides indispensable methods to build applications, like *getCurrentUserDir()*, which returns the path to the current user's home directory. For example:

```
$path = service('um','getCurrentUserDir');
```

Knowing the path to the home of the user that executed the application is useful to save files. Let's imagine, for example, we are creating a text editor and we want to save what the user wrote (using methods explained later) to a file called 'myFile.txt' inside of the “Documents” folder of the user's home:

```
$path = service('um','getCurrentUserDir'); //We get the user's home dir
$path .= 'files/'; //We add files/ since user's files are in this folder
$path .= 'Documentors/myFile.txt'; //We add the rest of the path to the file
service('vfs','create',array($path));
```

With this code, we have created the empty file. For that process, we needed to obtain the path to the user's home directory to create the file.

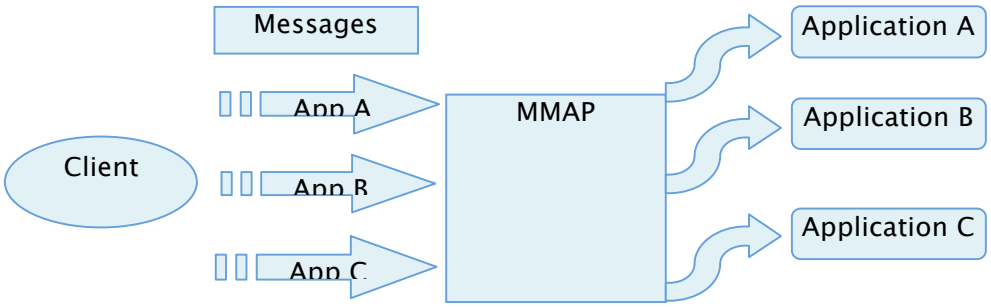
2.3 MMAP Message Mapping

Before introducing MMAP, we must first get to know the communication between the client and the server.
The eyeOS applications reside in the server, storing and processing the data in it, being for the client the medium to visualize and interact with the application. For example, the client is responsible of displaying a window, or sending a message to the server indicating a button has been clicked.

The communication between the client and the server is done via messages. A message is a petition made by the client to the server, sending or asking certain information.

Now that we know what a message is, we can move on and explain how MMAP takes part in this process.

MMAP is responsible for routing every received message in the server to the applications. A good comparison could be made with a postal system. When a mail arrives to the post central, it is classified according to its addressee, and finally it is delivered by the postman. MMAP would do the parts of organizing and delivery, since it classifies and transmits the message to the applications.



As a final note, you must know that MMAP is an automated service: it does not require any interaction with the developer at all. Nevertheless, it is important to know how to receive its messages, and that is explained in chapter 4.

2.4 PROC

PROC is responsible for the process managing, and provides methods to launch, end, list processes, etc.

When an application is launched with PROC, two very important variables for the application development are defined: **myPid** and **checknum**.

myPid is a unique 4 characters length numerical value that identifies every process and **checknum**, just like **myPid**, is a unique number, although it is 8 characters length and identifies every process in the client-server communication.

In this application, application “app1” executes the application “app2” and then it terminates itself.

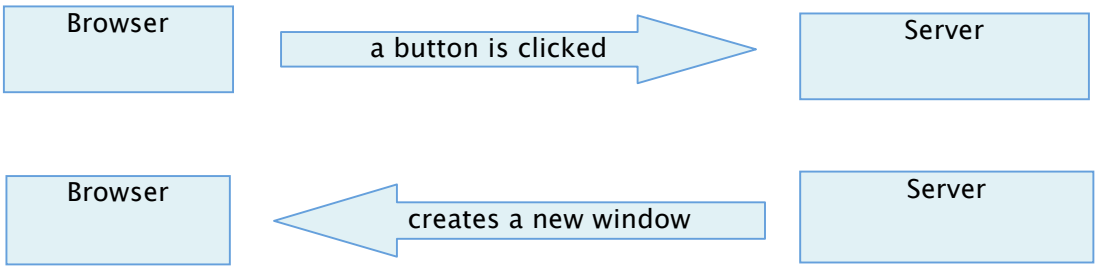
```
global $myPid;   //Pid of application “app1”
$pid = $myPid;   //We store the pid in another variable
service('proc','launch',array('app2');       //We call launch, indicating the name of the application to open
service('proc','close',array($pid);//We call close indicating the pid of app1
```

PROC provides a wide variety of process managing methods, not limited to launching or closing applications. For example, the *findPidByName()* function, locates a given process with specified **pid** and returns the name of its application name.

2.5 eyeX

As said in chapter 2.3, the mmap service manages the messages coming from the client. All messages have a response in XML format, which is interpreted by the browser.

The response XML contains basic orders to modify the interface of the client. For example, we create a button that creates a new window when it is clicked:



In the first case, the message goes from the browser to the server. In the second case, the response to the message includes the orders for the browser to create a new window.

The eyeX service is the responsible for managing the response XML sent by the server. The applications request eyeX basic operations, such as creating windows or showing an image, and it is eyeX that transforms those orders in an XML message.

For example, we can use eyeX to create a message box in the browser, just calling eyeX with *service()*:

```
service('eyeX','messageBox',array('content'=>'This is a message'));
```

2.6 Extern, external file manage

The extern service allows the client to download files from the server. Those files must be located inside the extern/ directory. (see chapter 1.6).

Extern allows to create URLs accessible from the outside, allowing to host images, CSS files, JavaScript files, etc.

Its working is very simple: if we want to get a file called style.css inside of a directory called 'test' located in eyeOS/extern/ we just should use the URL:

```
index.php?extern=test/style.css
```

This service exists to allow the developers to use images, stylesheets and other files in their applications, without worrying about the eyeOS/ directory is not visible from the outside.

2.7 Log

Log does not require any interaction with the developer. It works autonomously registering the system's activity to log files, annotating the time and user of every action. Its presence does not reverberate at all at the time of building applications.

2.8 Sec

Sec is another autonomous system service, although it reverberates on the developers, unlike the log service.

Sec is responsible of making eyeOS run in a secure environment. To do that, it disables PHP's magic_quotes and register_globals when initiating.

Thus it's not necessary to worry whether these options are enabled or not: you will always see them disabled.

3. System libraries

As we saw in chapter 1.5, the libraries are collections of functions we can use from our applications to make the development easier.

In eyeOS there are libraries for a wide range of tasks, although there are some very important ones, such as eyeWidgets, eyeXML, i18n and errorCodes.

In this chapter we will study how those libraries work and how we can use them in our applications.

3.1 errorCodes

When parts of our application are delegated to third parts (services, libraries, etc.) it is compulsory to standarize the error reporting so we can identify the errors ocurred in any moment. The errorCodes library exists to cover this need, implementing a unified and extensible method to report errors.

This library provides two functions: **setErrorCode** and **getErrorCode**.

setErrorCode() is used to store a value that indicates the type of error (normally represented by a number).

getErrorCode() retrieves the error previously stored by setErrorcode so that it can be identified.

All system services have values of errors reserved and defined as a constant. Each one of these value is called an “errorCode”.

Some errorCodes (you can check the documentation pages for all of them):

EYEOS_LOG_BAD_VALUE = -404

EYEOS_LOG_SERVICE_IGNORED = -401

In this example, we can see how the errorCodes library is used to handle a common action:

```
$path = '/etc/shadow';   //File's directory
$fp = service('vfs','open',array($path));   //We try to access the file
//If VFS returns false, we move on to examine which error code was set
        if($fp == false) {
        if($fp == VFS_INSUFFICIENT_PERMISSIONS) {
//We display a message box showing what type of error occurred
                service('eyeX','messageBox',array('content'=>"You do not have enough permissions."));
        }elseif($fp == VFS_FILE_NOT_EXISTS){
                service('eyeX','messageBox',array('content'=>"The file does not exist."));
        }else{
                service('eyeX','messageBox',array('content'=>"An unknown error occurred."));
        }
        return false;
}
```

In the example, we are trying to open a file that in Unix systems normally is only accessible by the administrator, and that in Windows systems it does not exist.

As you can note, in both cases we have a lot of possibilities the file is not accessible, and that is the reason we use errorCodes to discover the exact type of error and report it.

3.2 eyeXML

The XML format is widely used in eyeOS, so much that it would even be considered one of its pillars. For example, it is used to store configurations, transport data and messages, among much other uses. Nevertheless, the use eyeOS gives to XML differs from the usual form, using the arrays as XMLs, and the XMLs as arrays. To make this model possible, two main functions are used, called *array2xml()* and xml2array.

Don't be frightened if it sounds a bit abstract; we provide an example to show how simple it is:

```
//We have a variable storing XML with information about a user called main, and we want to return its creation date using eyeXML's methods
$myXml = "
<eyeUser>
        <username>main</username>
        <password>4f5fba03a86607a215fe91bd47735689</password>
        <email></email>
        <createDate>20/02/2007</createDate>
        <group>phpCoders</group>
</eyeUser>";
//Now, we transform the XML to an array using xml2array()
$myArray =reqLib('eyeXML','xml2array',array($myXml));
//We only return the value of the field createDate
return $myArray['eyeUser'][0]['createDate'][0];
```

The XML is transformed to the following array:

```
Array
(
    [eyeUser] => Array (
        [0] => Array(
            [username] => Array(
                [0] => main
              )
            [password] => Array(
                [0] => 4f5fba03a86607a215fe91bd47735689
              )
            [email] => Array(
                [0] =>
```

```
                )
            [createDate] => Array(
                [0] => 20/02/2007
                )
            [group] => Array(
                [0] => phpCoders
                )
        )
    )
)
```

eyeXML also provides functions to open and write XML files easily (with the methods getXMLfile and setXMLfile) and to directly handle XML configuration files (with methods getXMLconfig and setXMLconfig).

3.3 i18n Internationalization

The internationalization system is integrated in the eyeWidgets library (see chapter 3.4) and it works almost automatically. Nevertheless, in some occasions it is necessary to add manually the feature, specially in cases in which the Toolkit is not used such as some special visualization in the client (for example, the use of methods rawjs and runjs).

To implement the internationalization automatically is as easy as making a call to the *translate()* function before sending any text to the client. We need to bear in mind that the text to translate **must be written in English**.

```
$myText = "Hello World!"; //Our original text must always be written in English

$myTextTranslated = reqLib('i18n','translate',array($miTexto));  //We translate the text

$js = "alert('$myTextTranslated');"; //Javascript code using the translated text

service('eyeX','rawjs',array('js'=>$js));  //We send the rawjs order with our Javascript code
```

In this example, we see how to translate a given static text. In most occasions this method is sufficient for most cases, but there may be some situations in which the use of this method could not be appropiate, such as with dynamic texts. i18n is also capable of translating these kind of texts. Let's see an example:

```
$myText = "Hi %s, my pid is %s";//We replace with an %s the username and the pid

$tokens = array($currentUser,$myPid);
$myText = reqLib('i18n','translate',array($myText,$tokens));  //We translate our text with the tokens
//The resulting text translated in Spanish would be: "Hola root, mi pid es 4182"
```

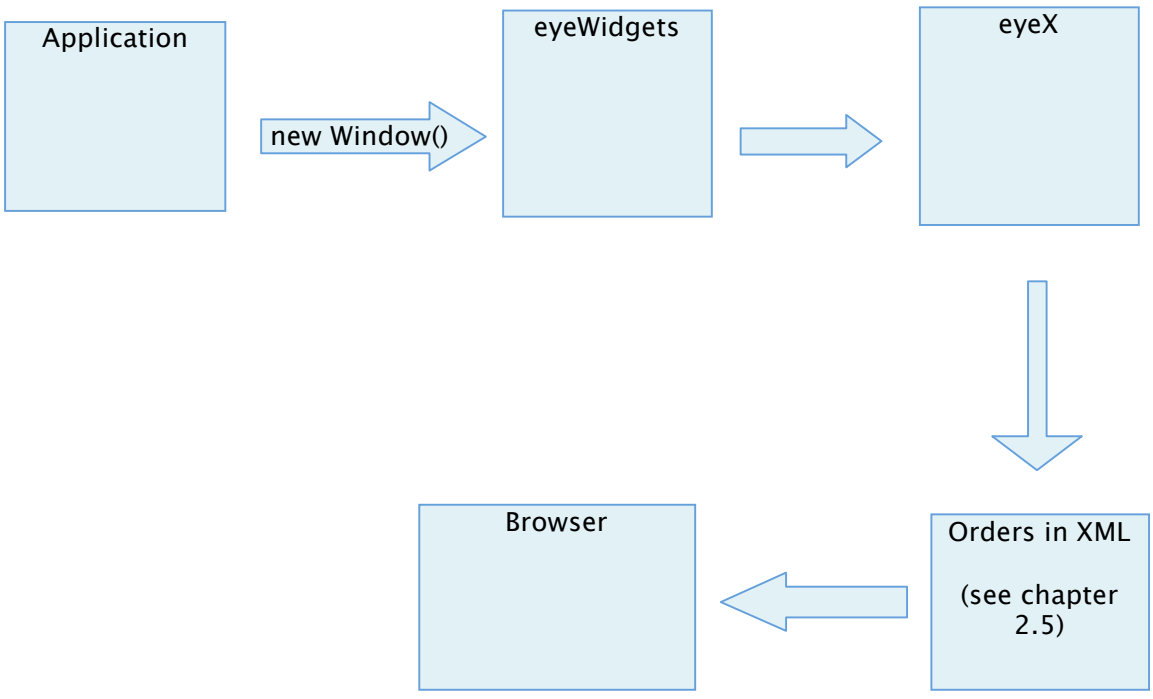
As the example shows, translating a text with dynamic data is easy: we only have to replace that data with "%s" and provide the function an array with every dynamic data in it.

3.4 eyeWidgets

The eyeWidgets library provies a quick form of creating graphical interfaces in eyeOS.
This library provides a set of classes used to create and modify visual elements in applications.
Among other uses, it allows to create windows, buttons, images, texts, tables, etc.
For example, creating a new window in our application is as simple as typing:

```
$myWindow = new Window(array(
        'name'=>'Window1',
        'father'=>'eyeApps',
        'cent'=>1,
        'width'=>600,
        'height'=>500,
        'title'=>'Test Window'
));
$myWindow->show();
```

This way, with eyeWidgets we can create graphic elements a user can interact with.
eyeWidgets works with eyeX (see chapter 2.5), telling it to append to its response (the response ultimately sent to the client as XML) the necessary orders to draw windows, buttons, or other widgets.



From this point of view, eyeOS is a graphical Toolkit, similar to local toolkits such as QT, but with the difference it works via web.

4.How an application works

4.1 Preparing the environment

All projects must be developed in an adequate environment. If, for example we are developing an application for GNU/Linux, it is better to use a GNU/Linux environment, otherwise the task would become quite a bothersome task. Let's see which requirements our environment should have:

- **Subversion Client**: we will need it to download the latest eyeOS code.
- **Apache2 Server**: the only server officially supported.
- **PHP 5**: the required version since eyeOS 1.2 and above.
- **PHP Editor**: any suitable text editor is valid.

Our ideal environment is a personal preference: not everybody feels comfortable with the same applications. The eyeOS team uses all the same environment, which we thoroughfully recommend:

- Official Subversion Client: http://subversion.tigris.org
- IDE Eclipse PDT.
- A Unix-like OS: OpenBSD, FreeBSD, GNU/Linux etc.
- Mozilla Firefox 2.X.
    - Firebug Extension.
    - ANEC (Add n edit cookies) Extension.

4.2 Downloading the code

The eyeOS code is in a Subversion (our Version Control system) server, and it is organized as follows:

- **trunk**: contains stable and secure code, and it is the main eyeOS working branch.
- **branches**:
    - **unstable**: contains applications that do not match the eyeOS quality standard.
- **tags**: contains the freezed code for each version released.

The SVN server can be found at http://svn.eyeos.org.

4.3 Structure of an application

One important thing you should know to manage in eyeOS is the structure of an application. The directory and files tree of an application is as follows:

- eyeOS/apps/ -Directory that contains the PHP code of each application
- eyeOS/apps/**Application**/ -Application's main directory

- eyeOS/apps/**Application**/**apps.eyecode** –Initializing and ending code of an application
- eyeOS/apps/Application/events.eyecode –Event reception code
- eyeOS/extern/apps/Application –Main directory for the external resources of an application

The files and directorios marked in bold are required for an application to work properly.
We will focus to explain the use of every file and directory in the next sections.

4.4 Initializing an ending of an application

The initialization and ending of an application is contained in the apps.eyecode file.

This file contains two functions:

- **ApplicationName_run**: function called by PROC when launching an application.
- ApplicationName_end: function called by PROC when terminating an application **in case it exists**.

The only required function in this file is the _run function. Its use varies enormously depending of the kind and size of the application. A graphical application normally uses it to initialize the User Interface, but non-graphical and small applications concentrate all their code in this function.

4.5 Events

An event is described as an interaction by the user with the interface of our application. For example, when a user clicks a button an event is produced. The generated information of every event is sent to the server in the form of a message. As we explained above, the responsible part of treating those messages is the MMAP service.

MMAP is able to deliver every message thanks to the fact that each message contains the checknum of the addressee application and the name of the event. To deliver the message, MMAP finds a file named events.eyecode in the directory of the addressee application and tries to call a function named ApplicationName_on_EventName, passing as an argument the body of the message. An example of a valid event function would be:

```
function eyeNotify_on_reciveNewMessage($params){
        if(!$params) {
                return false;
        }
        service('eyex','messageBox',array('content'=>'New message recived'));
}
```

- **eyeNotify**: name of the application to which the message is addressed.
- **reciveNewMessage**: name of the event received.
- **$params**: parameters sent from the client.

4.6 Extern

Using extern in our application is very easy. We only have to place the file we want to access in extern/apps/ApplicationName/ and then use a URL to the file as stated in chapter 2.6.
The use of extern in an application is quite useful since it allows to load icons, CSS files, images, etc.

4.7 Explained Example

Now that we know how the files and functions affect an application's working, it is time to see an example:

apps.eyecode:

```
//Initializing function that created the interface
function eyeBasic_run($params=null) {
	//Window widget
	$myWindow1 = new Window(array(
		'name' => 'eyeBasic_WND',
		'father' => 'eyeApps',
		'cent' => 1,
		'width' => 250,
		'height' => 150,
		'title' => 'Example eyeApp'
	));
	//We draw the window now that it is defined
	$myWindow1->show();
	//Button widget
	$myButton1 = new Button(array(
		'name'=>'eyeBasic_BTN',
		'father'=>'eyeBasic_WND_Content',
		'caption'=>'I'm a basic eyeApp. Click me!',
		'x'=>40,
		'y'=>80,
		'signal'=>'buttonPress'
	));
	//We draw the button
	$myButton1->show();
}
//Ending function: it removes the interface
function eyeBasic_end($params=null) {
	eyeWidgets('unserialize');
}
```

As we can see, the initializing function of the application only creates the interface. It is not time

to explain how the eyeOS Toolkit works right now (that part is covered in chapter 5), however it is important to notice in the parameter called 'signal' in the button widget.

events.eyecode:

```
//Function that receives the event generated when the button is clicked
function eyeBasic_on_buttonPress($params="") {
	//We show a message indicating that the button has been pressed
	service('eyex','messageBox',array('content'=>'The Button has been pressed!'));
}
//Function that receives the event generated when the window is closed
function eyeBasic_on_Close(){
	//We make the application terminate itself
	service('proc','end');
}
```

The event receiver functions have a characteristic syntax as explained earlier, and are always called by MMAP.
As we can see, the name of the event 'buttonPressed' matches the content of the signal parameter of the button. The details about the Toolkit will be explained in chapter 5.

5. The eyeOS Toolkit in a nutshell

The Toolkit is the main part of an application in eyeOS. It is the library that helps programmers managing and creating graphical interfaces.

The basic element of this Toolkit is the window, inside of which other elements are created. All visual elements of the Toolkit, such as buttons, bars, boxes or windows among other are designated as widgets.

5.1 How the Toolkit works

Every visual element of an application has a class that represents it: in other words, every widget is a class.

With this model, adding a widget in our application is as easy as instantiating a class.

All the classes of the Toolkit take as an argument an indexed array when they are instantiated. This array contains parameters shared in common with other widgets and specific parameters for each widget.

The common parameters of all widgets are:

name – the name given to the widget, such as 'window1', 'button2', etc. This name will be used to refer to the widget in the application.

father – the father of the widget. In other words, the widget inside of which this widget is placed. If, for example we create a window in our application and want to create a button inside the window, we must indicate the name of the window as the father of the button. The background is a layer called eyeApps, which is the father of all windows that are not placed inside another window.

- **X** – the horizontal coordinate with regard to its father.
- **Y** – the vertical coordinate with regard to its father.
- **Horiz** – this flag specifies the form in which the horizontal (X) coordinates are taken. If it is set to 0 (the default value) the pixels are counted from the left. If set to 1, the pixels are counted from the right.
- **Vert** – exactly like Horiz, but with the vertical (Y) coordinates.

  The use of horiz and vert is justified: we may want to stick some element to the right or down border of a window, and those flags eliminate the need to calculate the size of the window to place the element.
- **Cent** – this key is used to center widgets. When it is set to 0 (the default value) nothing is changed. If it is set to 1, it makes the horizontal and vertical coordinates (X and Y, respectively) given be ignored, and centers the widget at the center of the father. If set to 2, the widget is centered horizontally, and when it is set to 3 it is centered vertically.

Aside from these parameters in the array, each widget has its own specific parameters. We will se

an example using a window widget to see this

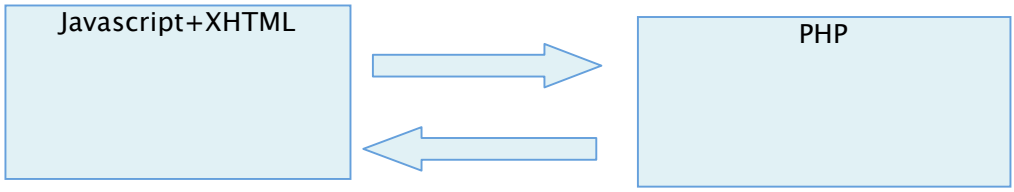
```
$myWindow = new Window(array(
        'name'=>'Window1',
        'father'=>'eyeApps',
        'cent'=>1,
        'width'=>600,
        'height'=>500,
        'title'=>'Example1',
));

$myWindow->show();
```

This example creates a window with a 600x500 pixels size, centered in the screen and with a title called 'Example1'.

5.2 The JavaScript side and the PHP side

Since eyeOS is executed in the server using PHP and it is visualized in the client, using XHTML and Javascript, it is necessary a mechanism to communicate both parts.

An eyeOS application could be defined this way:



The PHP side is where the classes reside, and in the Javascript side there is a system to draw the widgets, although the programmer does not need to deal with it.

To communicate both sides, Messages are used. They normally consist of Ajax calls that are made by the system when the user interacts with some part of an application. For example, if the user clicks a button, it emits a message, an Ajax call to the server, telling it the button is pressed.

The most interesting part of this process is that it occurs automatically, and the programmer only has to worry about creating the widgets and managing the events in a free way.

5.3 Messages

The messages in eyeOS are the way to call the events of an application. Those events, normally being Ajax calls to the server have 3 parameters (although the programmer does not need to deal with it, since it is automatically handled by eyeOS).

The first parameter is the checknum, a unique random number assigned to every application that lets eyeOS know to which application a message is directed to.

The second parameter is the signal, which is the name of the event of the application. As said

above, with each message received, eyeOS tries to execute a function from the events.eyecode file called applicationName_on_signal.

Most widgets capable of sending messages use their name as the signal to send. However, their constructor allows to specify a key named 'signal' to set the name of the signal sent to a desired value.

```
$myButton = new Button(array(
        'name'=>'Button1',
        'signal'=>'doSignal'
        'caption'=>'Click me!',
        'father'=>'Window1',
        'x'=>30,
        'y'=>30
));
```

This example creates a button with the text “Click me!”, that will produce the event 'doSignal' when clicked.

In addition, the messages have a third param, which is their value. This value is the content of the widget that produces the event. For example, a textbox that sends a signal when you enter the text on it would send its text. This parameter lets the PHP side to stay coordinated with the values of the Javascript/XHTML side.

5.4 Serialization

In eyeOS, the serialization consists in storing all the widgets of an application in the user's session so that the widgets remain available when an event is triggered. Let's see an example:

We create a textbox and a button, and we want to change the text of the textbox to the string 'Hello, World!' every time the button is clicked.

```
        $myTextbox = new Textbox(array(
                'name'=>'textBox',
                'width'=>212,
                'father'=>'Window1',
                'x'=>30,
                'y'=>50
        ));

        $myTextbox->show();


        $myButton = new Button(array(
                'name'=>'button1',
                'caption'=>'Change text',
                'father'=>'Window1',
                'x'=>120,
```

```
            'y'=>90,
            'signal'=>'change'
        ));
        $myButton->show();
```

Now we have the two elements drawn in the window (named Window1). Now we make that when the message 'change' is received, the event of the events.eyecode file changes the text inside the textbox:

```
function ejemplo1_on_change($params=null) {
        $textBox->setText('Hello, world!');
}
```

Since we gave the textbox the name 'textBox' when we created it, we can access this textbox just by referring it with this name. This way, we can access and modify the textbox using its class methods.

Serializing affects the memory of the server. We may be interested in drawing widgets without serializing them. To avoid serializing a widget is as easy as passing the number 1 as the argument of the *show()* method of the widget:

```
        $myTextbox = new Textbox(array(
            'name'=>'textBox',
            'width'=>212,
            'father'=>'Window11',
            'x'=>30,
            'y'=>50
        ));

        $myTextbox->show(1); //This widget is not serialized! We won't be able to access it from the events
```

5.5 Friends

The friends concept in eyeOS is a way to tie widgets so that an event on a widget causes another widget to refresh its content.

As we saw above, when a widget produces an event, it includes in the message its new content so that the PHP can have its most recent content.

However, we may want the widget to update the content of other widgets when producing an event. For example, in an application with a button and a textbox, we want to retrieve the content of the textbox when the button is pressed.

To update the content of that textbox when the button is pressed, we must make the textbox a friend of the button. Let's see an example:

```
	$myTextbox = new Textbox(array(
		'name'=>'textBox',
		'width'=>212,
		'father'=>'Window1',
		'x'=>30,
		'y'=>50
	));

	$myTextbox->show();


	$myButton = new Button(array(
		'name'=>'button1',
		'caption'=>'Change',
		'father'=>'Window11',
		'x'=>120,
		'y'=>90,
		'signal'=>'change'
	));
	$myButton->addFriend($myTextbox); //Now the textbox is a friend of the button
	$myButton->show();
```

This way, in the events.eyecode file, we can retrieve the content of the textbox this way:

```
function example_on_change($params=null) {
	$text = $textBox->text;
}
```

Now **$text** contains the text of the textbox at the moment of clicking the button.

6. Groups

Every user is member of at least one group, normally the public group. Each group has a folder with its name inside the groups/ directory where all the shared files between the group members are stored.

Currently there are few functions for group managing, although both the UM and VFS services offer some interesting ones:

- **um_getCurrentGroups**: returns a list of all existing groups.
- **vfs_real_getDirContent_group**: returns the contents of the folders of a group.

The files stored in a group do not have any type of restriction to the other members. For example, user A can delete a file of user B in the group.

7 Aliases

The aliases in eyeOS are used to shorten the syntax of certain system calls, such as services or libraries calls.

All the aliases can be found in system/kernel/alias.eyecode. Not all libraries have an alias, only the most used ones have aliases. Some of the most used aliases are:

- **vfs($call,$arg)**: alias of the function service('eyex',$call,$arg)
- **eyeXML($call,$arg)**: alias of the function reqLib('eyeXML',$call,$arg)

A code using aliases is much clearer, so the use of aliases is encouraged.

8 Listener Service Calls

There are certain cases in which it is necessary to make our application know when other applications make service calls. A clear example of this necessity would be a process monitoring application. To solve this need, the eyeOS kernel provides the listener service calls feature.

A listener service call (LSC from now on) is just a notification from the kernel to our application that a call to the function X of service Y has been made. To make this notification the kernel uses a similar method to that of MMAP, but this time using a file called com.eyecode.

When we register a LSC, we indicate the kernel 3 parameters: a function, a service and an identifier. The identifier is used to call the function with the name applicationName_com_identifier in the file com.eyecode of the registrar application. An example would be:

```
addListenerServiceCall('onKill','proc','close',1);
```

- **onKill**: the identifier
- **proc**: service of the function to listen
- **close**: function to listen
- **1**: specifies that the LSC must be called after the execution of the function

This example adds a listen to the call of the function “close” of the service “proc” and would call the function applicationName_com_onKill from the file com.eyecode after performing the service call.

9 Full Screen

Currently, this Toolkit does not have capability to make applications in full screen mode. Nevertheles it is easy for an application to use this mode making the following changes:

1 – Obtain the width and height of the session screen (variable $_SESSION['SCREEN']).
2 – Create a window with the obtained width and height, specifying the window has a FIXED_WINDOW type.
3 – Send the following Javascript code:

```
//We move the content of the window at the top
eyeX('rawjs',array('js' => 'document.getElementById("'.$myPid.'_windowName_Content").style.top="0px";'));
//We lower the z-index of the fixed elements of the desktop (the default value is 10000)

eyeX('rawjs',array('js' => 'document.getElementById("minimizedAppsLeft").style.zIndex="0";'));
eyeX('rawjs',array('js' => 'document.getElementById("minimizedAppsRight").style.zIndex="0";'));
eyeX('rawjs',array('js' => 'document.getElementById("minimizedApps").style.zIndex="0";'));
```

This way, the created window will not use borders, and will appear in full screen mode. An important issue is that the application is responsible of **restoring** the elements of the desktop to the previous state.
Note that this feature is a hack, and must be always used carefully.

10 Message Tables

As explained in chapter 2.3, MMAP is responsible of addressing the messages, and to do that it calls functions that use the pattern *applicationName_on_eventName*. Normally, this system is enough to satisfy most of the programmers' needs to have a clean and well-organized code. However, you might want to skip that syntax, and that's why the message tables exist.

To use the message tables we simply must create an indexed array that relates the name of the event to the function we want to be called. Let's see an example applied to the example of chapter 4:

events.eyecode

```
$messageTable['buttonPress']['function'] = 'launchMessageBox';
function launchMessageBox($params="") {

        //We show a message saying the button has been pressed
        service('eyex','messageBox',array('content'=>'The Button has been pressed!'));
}
```

As it can be seen, the function called by the event is now launchMessageBox instead of eyeBasic_on_buttonPress.

11 Important eyeOS Libraries

11.1 eyeSockets

When it comes to sockets, PHP provides lots of function to manage them. Nevertheless, their use is sometimes unintuitive and unclear. To solve this problem, eyeOS provides the eyeSockets library.

eyeSockets provides a class that offers a clean, quick and efective programming when establishing and using a connection. In example, we have multiple functions to write and read with this class.

11.2 eyeURL

The eyeURL library provides a simple and solid class to make HTTP requests. This protocol is necessary to interact with multiple services, and at the same time some protocols are based on it, such as the XML-RPC protocol.
The uses of eyeURL are numerous and varied, providing eyeOS an infinity of possibilities, like for example file downloading, web services interaction, etc. Its use is very simple, let's see it with an example:

```
$httpClient = eyeURL('getHTTPClient');
//Direction URL
$httpClient->setURL('http://www.google.com/');
//We send the request

$httpClient->sendRequest();
//We only obtain the body of the response
$html = $httpClient->getResponseBody());
```

This example makes a request to the well-known web search service, and obtains the body of the response.

11.3 simpleZip

A powerful library to create ZIP files. Its syntax is very simple, and it can be very useful in our application if we want it to have the ability to generate ZIP files on the fly and offer them for download.

11.4 eyePear

One of the best qualities in PHP is its large community, and an example of this is the PEAR project (http://pear.php.net/).

PEAR is a framework and a distribution system for reusable PHP components. In eyeOS only the framework is used (the collection of its libraries), having to do few little modifications to totally adapt a library from PEAR to eyeOS.

All libraries from PEAR complement themselves, and at the same time they are based in a class called PEAR in which they delegate some actions, such as the error reporting. If we want to port a

PEAR library to eyeOS, we must follow only 3 steps:

- Provide the PEAR class to the libraries (eyeOS automatically does this task)
- Check and modify the include parts (the eyeOS structure differs)
- Create a wrapper that will be called by the eyeOS kernel

A good example of this is the eyeCrypt library, which implements two functions: crypt and decrypt, which are a wrapper of the PEAR classes.

12 Extras directory

The extras directory contains resources which are not integrated in the system, such as binary applications, external configuration files, etc.

A good example of this kind of resources in this directory would be the conversion between different formats, a task which is normally delegated to external programs.

13 How sessions work with eyeSessions

The eyeSessions library provides an abstracted method to obtain and save variables and arrays in the session. An example of its use would be:

```
//We check if the variable exists
if(eyeSessions('checkVar',array(TABLENAME)) == false){

        //If it does not exist, we create the array
        eyeSessions('makeArrayVar',array(TABLENAME));
}
```

Despite seeming redundant in respect to the use of the superglobal variable $_SESSION, the use of this library is really useful and clear when making intense use of the session.

14 Sharing your work with the eyeOS Community

eyeOS is an Open Source web operating system and applications platform, and aims to be the strongest free, open source web development platform out there.

This can only be possible with the work of thousands of people who do release they work as Open Source and share it with the rest of the eyeOS community. There are lots of available licenses out there. We recommend to use the GNU ones, but you are free to choose what's the best Open Source license for your work.

For this reason, we aim you to release your work (from your very first application to a full system of applications and system tweaks) in the eyeOS Applications Community:

http://www.eyeos-apps.org

You can also find help and updates about the system and its community in the official home page:

http://www.eyeos.org

15 About this manual

15.1 License

15.2 Authors and contributors

Creators

Jose Carlos Norte

Alejandro Fiestas

English translator

Daniel Gil

Special thanks to

Pau Garcia-Milà

Marc Cercós

Anaël Ollier

You can contact the eyeOS Team at **team@eyeos.org** .