System Workers

System workers allow JavaScript code to call any external process (a shell command, PHP, etc.) on the same machine. By using callbacks, Wakanda makes it possible to communicate both ways.

```
    // Windows example to get access to the ipconfig window
var myWinWorker = new SystemWorker("C:\\windows\\System32\\ipconfig.exe");
myWinWorker.wait(1000);

    // Mac OS example to change the permissions for a file on Mac OS
    // chmod is the Mac OS command used to modify file access
var myMacWorker = new SystemWorker("chmod +x /folder/myfile.sh");
```

Usually, system workers are called asynchronously, but Wakanda also provides you with the **SystemWorker.exec( )** method to handle synchronous calls.

You can also call "sandboxed" system workers by their name, using a specific definition file named *systemWorkers.json*. For example, if a "git" system worker is defined in this file, you would then be able to execute a git command by creating the following worker:

```
var worker = new SystemWorker ( "git version", null );
```

For more information, please refer to the (see **Configuring systemWorkers.json file**) section.

Detailed Example

The following example demonstrates how to use both the synchronous SystemWorker.**SystemWorker.exec( )** function as well as the asynchronous way using **SystemWorker( )** constructor. In particular, it will show how to write and read data to and from an external process.

The example compresses the content of a Buffer object using gzip. Then it decompresses it back using gzip again, and compares it to the original input (they should match).

```
    // Create input, make some "binary" data.
var input = new Buffer('abcde', 'ascii');
input[3] = 123;
input[4] = 250;

    // Use gzip to compress buffer. Note that when call without arguments,
    // gzip will compress data from stdin and ouput compressed gz file on stderr.
var result = SystemWorker.exec('gzip', input); // Synchronous call
if (result = null)
    debugger;     // Check gzip is at /usr/bin/gzip.

    // Decompress the gz file:
var worker = new SystemWorker('gzip -d'); // Create an asynchronous system worker

    // Send the compressed file on stdin. result.output is a Buffer object
    // containing binary compressed data (the gz file). Note that we call
    // endOfInput() to indicate we're done. gzip (and most program waiting
    // data from stdin) will wait for more data until the input is explicitely closed
worker.postMessage(result.output);
worker.endOfInput();

    // We expect to receive our initial input back. That input contains
    // binary data (input[3] == 123 and input[4] == 250).
worker.setBinary(true);

    // Prepare a Buffer to read back decompressed file. Then setup callback.
var buffer    = new Buffer(100);
var index     = 0;

worker.onmessage = function (obj) {

        // Note: Code of an actual product should perform bound checking.
    obj.data.copy(buffer, index);
    index += obj.data.length;
}

    // The SystemWorker is asynchronous, wait for termination. This will also
    // allow callbacks to be processed.
worker.wait();

// Check that input matches decompressed data.
var isOk    = true;

if (index != input.length) {
    isOk = false;
    console.log("Length doesn't match!\n");
}

for (var i = 0; i < input.length; i++)
    if (buffer[i] != input[i]) {
        isOk = false;
        console.log('Decompressed data doesn\'t match!\n');
        break;
    }

if (isOk)
    console.log('Ok.\n');
```

Configuring systemWorkers.json file

What is the systemWorkers.json file?

The **systemWorkers.json** file is a specific Wakanda Server settings file used to declare a list of available executable files that can be launched as system workers within your solutions. Each executable is referenced through a name, which can be passed as parameter to the system worker constructors, such as **SystemWorker( )** or SystemWorker.**exec( )**.

Using this file, your system workers can work in a sandboxed environment so that only relative file system references are ever passed to any system worker, and not absolute or relative system file paths.

Contents of a systemWorkers.json file

A *systemWorkers.json* file contains the following objects:

Object	Type	Description
"name"	String	Name of the system worker, that will be passed as parameter to the **SystemWorker( )** or SystemWorker.**exec( )** constructors
"executable"	Array	(optional) Each element contains an object with a single "path" attribute. The attribute value defines a local path to the executable file. Several paths can be defined for the same executable, paths are tested one by one in array order and the first one that exists on disk is used as the system worker's executable.

For example, if there is a "systemWorkers.json" file defined with the following content:

```
[
    {
        "name" : "git",
        "executable" :
        [
            { "path" : "/usr/bin/git" },
            { "path" : "/usr/local/bin/git" },
            { "path" : "/usr/local/git/bin/git" },
            { "path" : "C:\\Program Files (x86)\\Git\\bin\\git.exe" }
        ]
    }
]
```

You would then be able to execute a Git command by creating the following worker:

```
var myWorker = rnew SystemWorker ( "git version", null );
```

In this way, a developer does not need to know the exact binary location and the exact full file paths used in a command.

Note that a system worker can be referenced simply by its name if the executable binary is properly registered in the OS (PATH environment variable, Windows registry...). For example, the following definition will be valid if Git is installed and is included in the PATH environment variable:

```
[
    {
        "name" : "git"
    }
]
```

Where to locate a systemWorkers.json file?

A *systemWorkers.json* file is supported by Wakanda Server at three locations:

- In the **Resources** folder of Wakanda Server application package: this file describes default built-in system workers (such *git* or the prompt command for example). This file is provided by default and should not be edited.
- At the solution level: system workers defined in this location are private to the solution.
- At the project level: system workers defined in this location are private to the project.

When a *systemWorkers.json* file exists at several locations, each file is merged during execution.

Note: *You can also select a systemWorkers.json file when launching Wakanda Server through a command line (CLI) by using the* **system-worker <path>** *option, where <path> is the full pathname of the systemWorkers.json file. For more information, please refer to* **Administrating Wakanda Server (Unix)** *or* **Administrating Wakanda Server (Windows)** *section.*

SystemWorker Constructor

studio.SystemWorker( )

void **studio.SystemWorker**(String *commandLine* [, String | Folder *executionPath*])

Parameter	Type	Description
commandLine	String	Commande line to execute
executionPath	String, Folder	Directory where command is executed

Description

The **studio.SystemWorker( )** constructor method allows you to create and handle a SSJS *SystemWorker* type object from your extension code.
For more information about *SystemWorker* objects, please refer to the **SystemWorker Instances** description.

SystemWorker.exec( )

Object | Null **SystemWorker.exec**(String *commandLine* [, Buffer | String *stdinContent*[, String | Folder *executionPath*]])

Parameter	Type	Description
commandLine	String	Command line to execute
stdinContent	Buffer, String	Data to send on stdin
executionPath	String, Folder	Directory where command is executed
Returns	Object, Null	Resulting object

Description

The **SystemWorker.exec( )** method launches an external process in synchronous mode.
Under Mac OS, this method provides access to any executable application that can be launched from the Terminal.
***Note:** The **SystemWorker.exec( )** method only launches system processes; it does not create interface objects, such as windows.*

First syntax: SystemWorker.exec(commandLine[,stindContent[,executionPath]])

In the *commandLine* parameter, pass the executable application's absolute file path to be executed, as well as any required arguments (if necessary). Under Mac OS, if you pass only the application name, Wakanda will use the PATH environment variable to locate the executable. This parameter must be expressed using the System syntax. For example, under Mac OS the POSIX syntax must be used.
In the optional *stdinContent* parameter, you pass the data to send on the stdin of the external process (if any). You can use either a *string* or a *blob* parameter, both of which can be empty.
In the optional *executionPath* parameter, you can pass an absolute path to a directory where the command must be executed. This parameter must be expressed using the System syntax. You can also pass a *Folder* object.

Second syntax: SystemWorker.exec(commandLine, options)

With this syntax, in *commandLine* you pass the name of a system worker that has been previously defined in a *systemWorkers.json* file, as well as parameter(s). For more information about system worker definitions, please refer to the **Configuring systemWorkers.json file** section.
Pass in *commandLine* either a string that represents the line to execute, or an array of strings where the first element is a system worker name and the rest of elements are command line parameters.
For example:

```
SystemWorker.exec( "git version", null);
```

or

```
SystemWorker.exec( [ "git", "version" ], null);
```

Pass in the *options* parameter a set of attributes to configure the system worker execution. Pass *null* if you do not use this parameter. The following (optional) attributes are supported:

Atttribute	Type	Description
options.folder	Folder | String	Root folder for the worker executable. Native relative file paths will be resolved with this folder as parent. The '.' reference resolves to this folder.
options.parameters	Object	Allow to pass named parameters to command line. Named parameters are passed in curly brackets. *{name}* is replaced with the value of the *options.parameters.name* attribute. Parameters can be of type ***String***, ***File*** or ***Folder*** (see examples)
options.quote	String	Escape character (can be an empty string). Named parameters may need to be escaped depending on the system worker and OS on which it is executed (see examples)
options.stdin	String	Can be null or empty string. String data to pass to system worker at launch.
options.variables	Object	Can be null. Defines custom environment variables for the system worker.
options.kill_process_tree	Boolean	Pass this option with value **true** to terminate all processes launched by the system worker (process tree) once the parent process is terminated. Otherwise (**false** value or option omitted), the process tree is left untouched.

Resulting object

If the external process was launched successfully, **SystemWorker.exec( )** returns a resulting object. Otherwise, the method returns a **null** value.
If returned, the resulting object contains the following attributes:

Attribute	Type	Contents
exitStatus	Number	Integer value depending on the executable. If the executable considers the operation has been executed successfully, exitStatus value is 0
output	Buffer	stdout result of the command

error	Buffer	stderr result of the command

Note about returned data

Data returned by an external process can be of a very different natures. This is why objects receiving the resulting *stdout* can handle binary data:

- in the case of the **SystemWorker.exec( )** method, ***result*.output** and ***result*.error** are of the *Buffer* type
- in the case of the **SystemWorker( )** method, both **onmessage** and **onerror** functions can use **setBinary( )** to return binary data instead of UTF-8 strings.

By default, the Mac OS system uses UTF-8 for character encoding. But on Windows, many different configurations can be defined. Also, when executing a system command, it is usually recommended to use an instruction such as:

```
'cmd /u /c "<command>"'
```

This will return data in Unicode: the "/u" option asks the system to return Unicode characters; the "/c" option specifies the command to execute.

Example with the 'dir' command on Windows:

```
var result = SystemWorker.exec('cmd /u /c "dir C:\\Windows"');
result.output.toString("ucs2");
```

By default, *Buffer*.toString() converts data from UTF-8. If the source string is not correctly encoded, the result will be an empty string. You need to pay particular attention to data encoding when executing SystemWorker calls.

Using built-in commands

Keep in mind that **SystemWorker.exec( )** can only launch executable applications; it cannot execute instructions that belong to the shell (which is a command interpreter). In this case, you need to use an interpreter. Built-in **bash** or **cmd** commands (such as "dir", "cd" as well as "|" or "-") cannot be called directly in a SystemWorker.

For example:

```
macWorker = new SystemWorker("bash -c 'ls -l'");    // Mac or Linux
winWorker = new SystemWorker("cmd /c dir");        // Windows
    // The following statements do not work:
    // macWorker = new SystemWorker("ls -l");
    // winWorker = new SystemWorker("dir");
```

Default named system workers have been declared in the Wakanda Server *systemWorkers.json* file, so that you can execute shell commands directly using the name-based syntax. Here is, for example, how to list all environment variables on Windows:

```
SystemWorker.exec( [ "cmd", "/C", "set" ], null);
```

And here is how to list the content of a folder on OS X:

```
SystemWorker.exec( [ "sh", "-c", "ls -la /Volumes/MyDisk/MyFolder/MySubFolder/" ], null);
```

Example

The following Mac OS example creates a synchronous system worker that gets statistics from the server (and returns them):

```
function getLocalStats()
{
    // The object that will be returned
     var data = {};
     var isOk = true;

    // Function to get virtual memory stats
     function vm_stat() {

         var result = SystemWorker.exec('/usr/bin/vm_stat'); // launch external process in synchronous mode
         if (result != null) {
               var stdout    = result.output.toString();
               var lines    = stdout.split('\n');
               for (var i = 0, j = lines.length; i < j; i++) {
                    linedata = lines[i].split(':');
                    var key  = linedata[0].replace(/\"/g,'\"').replace(':','').trim().replace(/[\s-]/g,'_').toLowerCase
                    if (key != '' && parseInt(linedata[1]) > 0) {
                        data[key] = parseInt(linedata[1]);
                    }
               }
         } else {
            isOk = false;
            console.log('vm_stat failed to launch!\n');
         }
     }

     // Function to get cpu load at 1/5/15 minutes
    function iostat() {

        var result = SystemWorker.exec('/usr/sbin/iostat -n0'); // launch external process in synchronous mode
        if (result != null) {
            var stdout    = result.output.toString();
            var detail    = stdout.split('\n')[2].replace(/\s+/g,' ').trim().split(' ');

            data['user'] = parseInt(detail[0]);
```

```
            data['system'] = parseInt(detail[1]);
            data['idle'] = parseInt(detail[2]);
            data['one_mn'] = parseFloat(detail[3]);
            data['five_mn'] = parseFloat(detail[4]);
            data['fifteen_mn'] = parseFloat(detail[5]);

        } else {
            isOk = false;
            console.log('iostat failed to launch!\n');
        }
    }

    // Call sequence
    vm_stat();
    if (isOk)
        iostat();

    // Return the expected data if successful.
     if (isOk)
         return {
            date:               new Date(),
            cpu1mn:             data.one_mn,
            cpu5mn:             data.five_mn,
            cpu15mn:            data.fifteen_mn,
            mem_active:         data.pages_active,
            mem_inactive:       data.pages_inactive,
            mem_speculative:    data.pages_speculative
        };
    else
        return {};
}

var dumpAttributes = function (object) {

    var    k;
    for (k in object) {
        if (typeof object[k] == 'undefined')
            console.log(k + ': undefined');
        else
            console.log(k + ': ' + object[k].toString());
    }
}

// Display statistics for 10 seconds, refreshing each second.

var id;
var count = 0;
id = setInterval(function () {
    dumpAttributes(getLocalStats());
    if (++count == 10) {
        clearInterval(id);
    }
}, 1000);

// Process events for 15 seconds: Stats will be displayed for 10 seconds.
// Then wait 5 seconds before exiting the wait().

wait(15000);
```

Note: *For a similar example in asynchronous mode, see the example from the* ***SystemWorker( )*** *constructor.*

Example

Creating a named system worker and passing a command name attribute in the options parameter:

```
var options = { parameters : { command_name : "diff" } };
var result = SystemWorker.exec( "git help {command_name}", options);
```

Example

Creating a system worker using parameters, quote and variables options:

```
var fl = new File ( "/Project/Folder/File Name With Spaces.js" );
var options = { parameters : { file_ref : fl }, quote : '"' };
    //create an environment variable ENV_VAR_1 with value "value1"
options.variables = { ENV_VAR_1 : 'value1' };
var result = SystemWorker.exec( ["git", "status", "--", "{file_ref}"], options);
```

SystemWorker( )

void **SystemWorker**(String *commandLine* [, String | Folder *executionPath*])

Parameter	Type	Description
commandLine	String	Command line to execute
executionPath	String, Folder	Directory where command is executed

Description

The **SystemWorker( )** method is the constructor of the ***SystemWorker*** type class objects. It allows you to create a new ***SystemWorker*** proxy object that will execute the ***commandLine*** you passed as parameter to launch an external process. Under Mac OS, this method provides access to any executable application that can be launched from the Terminal.

Once created, a ***SystemWorker*** proxy object has properties and methods that you can use to communicate with the worker. These are described in the **SystemWorker Instances** section.

Note: The SystemWorker( ) method only launches system processes; it does not create interface objects, such as windows.

First syntax: SystemWorker(commandLine [, executionPath])

With this syntax, you enter file system paths directly as parameters. You do not use *systemWorkers.json* file(s).

- In the ***commandLine*** parameter, pass the application's absolute file path to be executed, as well as any required arguments (if necessary). Under Mac OS, if you pass only the application name, Wakanda will use the PATH environment variable to locate the executable.
- In the optional ***executionPath*** parameter, you can pass an absolute path to a directory where the command must be executed. You can also pass a ***Folder*** object.

Both ***commandLine*** and ***executionPath*** parameters must be expressed using the System syntax. For example, under Mac OS the POSIX syntax must be used.

Second syntax: SystemWorker(commandLine, options)

With this syntax, in ***commandLine*** you pass the name of a system worker that has been previously defined in a *systemWorkers.json* file, as well as a parameter. For more information about system worker definitions, please refer to the **Configuring systemWorkers.json file** section.

In ***commandLine***, pass either a string that represents the line to execute, or an array of strings where the first element is a system worker name and the rest of the elements are command line parameters.

For example:

```
SystemWorker( "git help merge", null);
```

or

```
SystemWorker( [ "git", "help", "merge" ], null);
```

In the ***options*** parameter, pass a set of attributes to configure the system worker execution. Pass *null* if you do not use this parameter. The following (optional) attributes are supported:

Atttribute	Type	Description
options.folder	Folder | String	Root folder for the worker executable. Native relative file paths will be resolved with this folder as parent. The '.' reference resolves to this folder.
options.parameters	Object	Allow you to pass named parameters to the command line. Named parameters are passed in curly brackets. *{name}* is replaced with the value of the *options.parameters.name* attribute. Parameters can be of the type ***String***, ***File*** or ***Folder*** (see examples)
options.quote	String	Escape character (can be an empty string). Named parameters may need to be escaped depending on the system worker and OS on which it is executed (see examples)
options.stdin	String	Can be null or an empty string. String data to pass to system worker at launch.
options.variables	Object	Can be null. Defines custom environment variables for the system worker.

Note about returned data

Data returned by an external process can be of a very different natures. This is why objects receiving the resulting *stdout* can handle binary data:

- in the case of the **SystemWorker.exec( )** method, ***result*.output** and ***result*.error** are of the ***Buffer*** type
- in the case of the **SystemWorker( )** method, both **onmessage** and **onerror** functions can use **setBinary( )** to return binary data instead of UTF-8 strings.

By default, the Mac OS system uses UTF-8 for character encoding. But on Windows, many different configurations can be defined. Also, when executing a system command, it is usually recommended to use an instruction such as:

```
'cmd /u /c "<command>"'
```

This will return data in Unicode: the "/u" option asks the system to return Unicode characters; the "/c" option specifies the command to execute.

Example with the 'dir' command on Windows:

```
var result = SystemWorker.exec('cmd /u /c "dir C:\\Windows"');
result.output.toString("ucs2");
```

By default, *Buffer*.toString() converts data from UTF-8. If the source string is not correctly encoded, the result will be an empty string. You need to pay particular attention to data encoding when executing SystemWorker calls.

Using built-in commands

Keep in mind that **SystemWorker( )** can only launch executable applications; it cannot execute instructions that belong to the shell (which is a command interpreter). In this case, you need to use an interpreter. Built-in **bash** or **cmd** commands (such as "dir", "cd" as well as "|" or "~") cannot be called directly in a SystemWorker.

For example:

```
macWorker = new SystemWorker("bash -c 'ls -l'");    // Mac or Linux
winWorker = new SystemWorker("cmd /c dir");        // Windows
    // The following statements do not work:
    // macWorker = new SystemWorker("ls -l");
    // winWorker = new SystemWorker("dir");
```

Default named system workers have been declared in the Wakanda Server *systemWorkers.json* file, so that you can execute shell commands directly using the name-based syntax. Here is, for example, how to list all environment variables on Windows:

```
SystemWorker.exec( [ "cmd", "/C", "set" ], null);
```

And here is how to list the content of a folder on OS X:

```
SystemWorker.exec( [ "sh", "-c", "ls -la /Volumes/MyDisk/MyFolder/MySubFolder/" ], null);
```

Example

The following Mac OS function creates an asynchronous system worker that gets statistics from the server (and returns them):

```
function getLocalStats()
 {
     // The object that will be returned
         var data = {};

     // Function to get virtual memory stats
         function vm_stat() {
             var myWorker = new SystemWorker('/usr/bin/vm_stat');
             myWorker.onmessage = function() {
                 var result = arguments[0].data;
                 var lines = result.split('\n');
                 for (var i=0, j=lines.length; i<j; i++) {
                    linedata = lines[i].split(':');
                    var key =   linedata[0].replace(/"/g,").replace(':','').trim().replace(/[\s-]/g,'_').toLowerCase();
                     if (key!='' && parseInt(linedata[1])>0) {
                         data[key] = parseInt(linedata[1]);
                     }
                }
                 exitWait(); // "unlock" the caller(s)
             }
        }

     // Function to get CPU load at 1/5/15 minutes
        function iostat() {
            var myWorker = new SystemWorker('/usr/sbin/iostat -n0');

            myWorker.onmessage = function() {
                 var result = arguments[0].data;
                 var detail = result.split('\n')[2].replace(/\s+/g,' ').trim().split(' ');
                 data['user'] = parseInt(detail[0]);
                 data['system'] = parseInt(detail[1]);
                 data['idle'] = parseInt(detail[2]);
                 data['one_mn'] = parseFloat(detail[3]);
                 data['five_mn'] = parseFloat(detail[4]);
                 data['fifteen_mn'] = parseFloat(detail[5]);

                 exitWait(); // "unlock" the caller(s)
             }
         }

     // Get the stats synchronously: Call a function, then wait until it calls exitWait()
         vm_stat();
         wait();
         iostat();
         wait();

     // Return the expected data
         return {
                 date               : new Date(),
                 cpu1mn             : data.one_mn,
                 cpu5mn             : data.five_mn,
                 cpu15mn            : data.fifteen_mn,
                 mem_active         : data.pages_active,
                 mem_inactive       : data.pages_inactive,
                 mem_speculative    : data.pages_speculative
         };
 }
```

***Note:** For a similar example in synchronous mode, see the example from the **SystemWorker.exec( )** constructor method.*

Example

This basic example on Windows creates an asynchronous SystemWorker that calls *netstat* and dumps the result using **console.log( )**:

```
    // Call netstat and make it refresh every 2 seconds
var systemWorker = new SystemWorker('c:\\Windows\\System32\\netstat.exe -n -p tcp 2');

    // Set up callback to display data from stdout
systemWorker.onmessage = function (obj) {
    console.log(obj.data);
}

    // SystemWorker termination event. It will make the script exit from wait().
```

```
systemWorker.onterminated = function () {
    exitWait();
}

    // Set a timeout. After 10 seconds, it requests termination of the SystemWorker.
    // Thus network data is displayed every 2 seconds for 10 seconds.

setTimeout(function () {
    systemWorker.terminate();
}, 10000);

// Asynchronous execution.
wait();

console.log("Done!\n");
```

Example

The following example changes the permissions for a file on Mac OS (*chmod* is the Mac OS command used to modify file access):

```
var myMacWorker = new SystemWorker("chmod +x /folder/myfile.sh");   // Mac OS example
```

Example

Creating a named system worker and passing a command name attribute in the options parameter:

```
var options = { parameters : { command_name : "diff" } };
var systemWorker = new SystemWorker( "git help {command_name}", options);
```

Example

Creating a system worker using parameters, quote and variables options:

```
var fl = new File ( "/Project/Folder/File Name With Spaces.js" );
var options = { parameters : { file_ref : fl }, quote : '"' };
    //create an environment variable ENV_VAR_1 with value "value1"
options.variables = { ENV_VAR_1 : 'value1' };
var systemWorker = new SystemWorker( ["git", "status", "--", "{file_ref}"], options);
```

SystemWorker Instances

System worker objects are created with the **SystemWorker( )** constructor method, which is available at the global application level. A *SystemWorker* object allows you to launch and control any external process on the server.

onmessage

Description

The **onmessage** property contains the function to call when a message is received from the external process.
The defined function will receive a single object as a parameter that has the following properties:

Property name	Type	Property value
type	String	"message"
target	*SystemWorker*	SystemWorker object which triggered the callback
data	String or Buffer(*)	Content of stdout

(*) see **setBinary( )** description.

Pay attention to the fact that the **onmessage** callback function can be called more than one time by the external process: the *stdout* can be sent in different parts by the external application, or the *stdout* buffer size (4096 bytes) can be overflowed during the data exchange.
Whatever the case, it is usually recommended to store the received *data* and to read it after the **wait( )** method.

onerror

Description

The **onerror** property contains the function to call when an error is received from the external process.
The defined function will receive a single object as a parameter that has the following properties:

Property name	Type	Property value
type	String	"error"
target	*SystemWorker*	SystemWorker object which triggered the callback
data	String or Buffer(*)	Content of stderr

(*) see **setBinary( )** description.

onterminated

Description

The **onterminated** property contains the function to call when the external process sent a termination message.
The defined function will receive a single object as a parameter that has the following properties:

Property name	Type	Property value
type	String	"terminate"
target	*SystemWorker*	SystemWorker object which triggered the callback
hasStarted	Boolean	True if the commandLine has been executed (path was correct)
exitStatus	Number	Exit status returned by the executed command *undefined* if hasStarted was False
forced	Boolean	True if the user called **terminate( )** *undefined* if hasStarted was False

endOfInput( )

void **endOfInput**( )

Description

The **endOfInput( )** method closes the input stream (stdin) of the external process.
This method is useful when an attempt to write in the stdin of the external process using the **postMessage( )** is blocked for some reason. A call to **endOfInput( )** will release the execution.

getInfos( )

Object **getInfos**( )

Returns Object Information about the system worker

Description

The **getInfos( )** method returns an object containing information about the *SystemWorker*.
The returned object will have the following properties:

Property name	Type	Property value
commandLine	String	Command line to execute

hasStarted	Boolean	True if the external process has started executing
isTerminated	Boolean	External process has terminated
pid	Real	(Mac OS and Linux only)(*) PID of external process

(*) The **pid** property is also available on Windows.

getNumberRunning( )

Number **getNumberRunning**

Returns	Number	Number of running system workers

Description

The **getNumberRunning( )** method returns the number of *SystemWorker* objects currently running on the server.

postMessage( )

void **postMessage**(String | Buffer *stdin*)

Parameter	Type	Description
stdin	String, Buffer	Input stream to write

Description

The **postMessage( )** method allows you to write on the input stream (stdin) of the external process.
Pass the string value to write in *stdin*.

The **postMessage( )** method also accepts a *Buffer* type value in *stdin*, so that you can post binary data.
You can use the **setBinary( )** method to make **onmessage** and **onerror** return *Buffer* values in the *data* member.

setBinary( )

void **setBinary**(Boolean *binary*)

Parameter	Type	Description
binary	Boolean	true = use binary data for messages, false = use string data (default)

Description

The **setBinary( )** method allows you to set the type of data exchanged in the *SystemWorker* through the **onmessage** and **onerror** properties.
By default, if this method is not called or if you pass **false** in the *binary* parameter, string type values are returned in the *data* parameter of the **onmessage** and **onerror** functions.
If you pass **true** in the *binary* parameter, *Buffer* type data will be returned in the *data* parameter. You can then use with this parameter all methods and properties of the **Buffer Instances** class.

terminate( )

void **terminate**([Boolean *waitForTermination* [, Boolean *killProcessTree*]])

Parameter	Type	Description
waitForTermination	Boolean	True to wait until the external process has terminated
killProcessTree	Boolean	True to kill the whole process tree

Description

The **terminate( )** method forces the external process to terminate its execution.
It you pass *false* to the *waitForTermination* parameter (or omit the parameter), the method will send the instruction to terminate and give control back to the executing script. If you pass **true** to the *waitForTermination* parameter, the method will send the instruction to terminate and block the executing script until the process has actually been terminated.

It you pass *false* to the *killProcessTree* parameter (or omit the parameter), the method will terminate only the parent process of the system worker. If you pass **true** to the *killProcessTree* parameter, the method will terminate the parent process as well as all process tree launched by the system worker.

wait( )

Boolean **wait**([Number *timeout*])

Parameter	Type	Description
timeout	Number	Waiting time (in milliseconds)
Returns	Boolean	True if external process has terminated

Description

The **wait( )** method allows you to set a waiting time for the *SystemWorker* to execute.
In *timeout*, pass a value in milliseconds. The *SystemWorker* script will wait for the external process for the amount of time defined in the *timeout* parameter. If you pass 0 or omit the *timeout* parameter, the script execution will wait indefinitely.
Actually, **wait( )** waits until the end of processing of the **onterminated** event, except if the *timeout* is reached,
This method returns *true* if the external process has terminated.
During a **wait( )** execution, callback functions are executed, especially callbacks from other events or from other *SystemWorker* instances.

Note: *The* ***wait( )*** *method is almost identical to the application* ***wait( )*** *method. You can exit from a* ***wait( )*** *by calling* ***exitWait( )****.*