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( )** constructor method for synchronous calls.

Detailed Example

The following example demonstrates how to use both the synchronous 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');
```

SystemWorker Constructor

exec( )

Object | Null **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	Null, Object	Resulting object

Description

The **exec( )** constructor method must be used with the SystemWorker( ) constructor to launch 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.*

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 MacOS the POSIX syntax should 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 as the execution path.

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
exitCode	String	Exit code of the command
output	Buffer	stdout result of the command
error	Buffer	stderr result of the command

Using built-in commands

Keep in mind that **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");
```

Note about returned data

Data returned by an external process can be of a very different nature. 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 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.

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
         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().r
                    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 proc
        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.*

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( )

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.*

- 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.

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

You can also pass a *Folder* object as the execution path.

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");
```

Note about returned data

Data returned by an external process can be of a very different nature. 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 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.

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().repl
                    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 **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');

    // Setup 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
```

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*])

Parameter	Type	Description
waitForTermination	Boolean	True to wait until the external process has terminated

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.

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( )**.*