# SSJS Modules

Wakanda proposes various SSJS utility modules for handling low-level, server-side features, such as TCP connections and events. These modules follow the CommonJS architecture.

# Events

Some SSJS objects can generate events, which are handled through callbacks functions (also called *listeners*). For example, *socket* objects created by the **net.Socket( )** or the **net.createConnection( )** method send events when the socket has established a connection.

All these objects implement the *EventEmitter* interface. An *EventEmitter* cannot be instantiated directly (it is implemented by objects, such as **net.Socket( )**, that receive events), but it is possible to define its own events.

You can access directly this module with the following statement:

```
requireNative("events");
    // temporary implementation
    // will be require("events") soon
```

## addListener( )

void **addListener**( String *event*, Function *listener* )

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |
| listener | Function | Function to execute when event is triggered |

### Description

*Note: This method does exactly the same thing as the method.*

The **addListener( )** method installs a new *listener* function to be called when the specified *event* is triggered by the object on which it is applied.

In *event*, pass the name of the event to trigger (whose name is case-sensitive). The events that are available depend on the emitter object.

For example, a *socket* object can generate the following events:

- "data": data is received in the socket
- "close": the socket is closed
- "connect": a socket connection is established successfully
- "error": an error occurred during the connection. A "close" event is always generated afterwards.

It is possible to define customized user events. You just need to call **addListener( )** with an *event* name of your choice; the *event* can then be triggered using the method.

Note that several listeners can be installed for the same *event*. In this case, all listeners are called in a first-in first-out basis. A listener array is maintained internally for each *event*.

### Example

See example for the constructor function.

## emit( )

void **emit**( String *event* [, Mixed *arg*,..., Mixed *arg*N])

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |
| arg | Mixed | Argument(s) to pass to the listener |

### Description

The **emit( )** method triggers the *event* for the object, optionally passing arguments to the listener(s).

Listener functions defined, for example, through are called and passed in *arg*, *arg2*,... as parameter(s).

This method is useful to trigger user-defined events.

## events.EventEmitter( )

void **events.EventEmitter**

### Description

The **events.EventEmitter( )** method is the constructor of objects in the EventEmitter class. Remember that you cannot instantiate *EventEmitter* objects directly; they are instantiated through the emitter itself (for example, a *socket*).

You can access the EventEmitter class using the following statement:

```
require('events').EventEmitter
```

All *EventEmitter* objects emit the 'newListener' event when new listeners are added.

## listeners( )

Array **listeners**( String *event* )

| Parameter | Type | Description |
|---|---|---|

| | | |
|---|---|---|
| event | String | Event name |
| **Returns** | Array | Array of listeners |

**Description**

The **listeners( )** method returns an array of listeners defined for the specified *event* in the object.

You can use this array to manage the listeners by, for example, removing one or more listeners.

## on( )

void **on**( String *event*, Function *listener* )

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |
| listener | Function | Function to execute when event is triggered |

**Description**

*Note: This method does exactly the same thing as the method.*

The **on( )** method installs a new *listener* function to be called when the specified *event* is triggered by the object on which it is applied.

In *event*, pass the name of the event to trigger (whose name is case-sensitive). The events available depend on the emitter object.

For example, a *socket* object can generate the following events:

- "data": data is received in the socket
- "close": the socket is closed
- "connect": a socket connection is established successfully
- "error": an error occurred during the connection. A "close" event is always generated afterwards.

It is possible to define customized user events. You just need to call **on( )** with an *event* name of your choice; this *event* can then be triggered using the method.

Note that several listeners can be installed for the same *event*. In this case, all listeners are called in a first-in first-out basis. A listener array is maintained internally for each *event*.

## once( )

void **once**( String *event*, Function *listener* )

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |
| listener | Function | Function to execute once when event is triggered |

**Description**

The **once( )** method sets a new *listener* function to be called only once when the specified *event* is triggered for the first time by the object on which it is applied. After the *listener* is called the first time, other events of this type are not triggered.

In *event*, pass the name of the event to trigger (whose name is case-sensitive). The events available depend on the emitter object. For more information, refer to the **addListener( )** method description.

## removeAllListeners( )

void **removeAllListeners**( [String *event*] )

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |

**Description**

The **removeAllListeners( )** method removes all the listeners of the specified *event* for the object to which it is applied. If the *event* parameter is omitted, the method removes all the listeners for the object.

## removeListener( )

void **removeListener**( String *event*, Function *listener* )

| Parameter | Type | Description |
|---|---|---|
| event | String | Event name |
| listener | Function | Listener function to remove |

**Description**

The **removeListener( )** method removes the specified *listener* from the listener array of the *event* for the object to which it is applied.

Note that the array indices of the following listeners are modified in the listener array.

## setMaxListeners( )

void **setMaxListeners**( Number *maxValue* )

| Parameter | Type | Description |
| --- | --- | --- |
| maxValue | Number | Maximum number of listeners per event, or 0 for unlimited |

**Description**

The **setMaxListeners( )** method defines the maximum number of listeners that can be added per event for the object to which it is applied.

By default, *EventEmitter* objects are limited to 10 listeners per event. This limitation can be helpful when looking for memory leaks. In some cases, you may want to change this maximum value. If so, pass a new maximum value to *maxValue* or zero to set an unlimited value.

# Mail ***Experimental***

The **Mail** module allows you to build an email message. Regarding this module, Wakanda is compliant with the RFC5322. Once prepared, the email can then be sent using the **send( )** method or with the help of the lower level **SMTP ***Experimental*** ** module methods.

The quickest way to build and send an email is to use the all-in-one **mail.send( )** method.

To invoke the "mail" module, you just need to execute the following statement:

```
var mail = require("waf-mail/mail");
```

An email message is made up of a header and a body. The header is a set of fields (such as **From**, **To**, etc.), some of which may appear several times. Only the "Date" and "From" fields are mandatory.

You can get and set fields by using the bracket [] or dot . syntaxes. For instance:

```
var mail = require("waf-mail/mail"); //to load the module
var myMessage = new mail.Mail();
myMessage.Subject = 'this is an email';
myMessage['To'] = 'somebody@somewhere.com';
```

Since some fields (such as "Comments") may appear an unlimited number of times, the value of a field can be an array of strings. It is preferable to use **addField( )**, **removeField( )**, and **getField( )** functions instead of direct assignments, because they provide error checking regarding field names and multiple fields are handled automatically. If values result in long lines, they must be folded accordingly (see section 2.2.3 of the RFC5322 specification).

The content of an email must be formatted according to section 2.3 of the RFC. The *Mail* object always stores properly formatted bodies. You can use the **setBody( )** and **getBody( )** to set or get formatted bodies. You can also use the **setContent( )** and **getContent( )** functions to handle "unformatted" bodies; they will do the appropriate conversions.

The **parse( )** function allows you to read an email from POP3 or IMAP responses.

## From

### Description

The **From** property contains an email address indicating who originally sent the message. Addresses in the From header are visible to the recipients of the message.

## Subject

### Description

The **Subject** property contains a text value concisely describing the topic covered in detail by the message body.

**Warning:** Usually, the subject of the message should not contain characters with diacritical marks (such as é, ö, etc.).

## To

### Description

The **To** property contains one or more complete email addresses indicating recipient(s) of the email. All the addresses identified in the "To" header will each be sent an original copy of the message. Each recipient of the message will see any other email addresses the message was delivered to.

## addField( )

void **addField**( String *name*, String *value* )

| Parameter | Type | Description |
|---|---|---|
| name | String | Message field name |
| value | String | Message field value |

### Description

The **addField( )** method adds a field definition to the *Mail* object.

Pass the name of the field in the *name* parameter and the value to set in the *value* parameter. Note that it is impossible to define field names having the same names as object's methods.

It is recommended to use this function rather than a direct assignment using the bracket [] or dot . syntaxes, because if the field is already defined, it will be automatically converted to a valid array of values. When you set a field directly using the bracket [] or dot . syntaxes, there is no check, and the previous value (if any) is just overwritten.

*Note: Some fields should not appear more than once. The addField( ) method currently does not check the uniqueness of a field. Note as well that the value parameter is not checked for syntax correctness.*

## getBody( )

Array **getBody**( )

| Returns | Array | Body of the message |
|---|---|---|

### Description

The **getBody( )** method returns the *body* of the *Mail*.

The returned body is an array of string lines that can be passed to SMTP methods to be sent (you just need to add a CRLF sequence at the end of each line).

## getContent( )

Array **getContent**( )

| | | |
|---|---|---|
| **Returns** | Array | Body content of the message |

### Description

The **getContent( )** method returns the formatted body (byte-stuffing removed) of the *Mail*.

The method returns an array of strings, each one representing a line of the body.

## getField( )

String | Array **getField**( String *name* )

| Parameter | Type | Description |
|---|---|---|
| name | String | Field name |
| **Returns** | Array, String | Value(s) of the field |

### Description

The **getField( )** method returns the current value of the field designated by *name*.

Note that an array of values will be returned if the field has been defined several times.

## getHeader( )

Array **getHeader**( )

| | | |
|---|---|---|
| **Returns** | Array | Header of the message |

### Description

The **getHeader( )** method returns the header of the *Mail* in the form of an array of strings.

The returned array is the whole contents of the header. You just need to add CRLF at the end of each element to send the *Mail* using SMTP methods.

*Notes: There are (currently) no general email rules checking. Keep in mind that "From" and "Date" fields are mandatory. Some fields may appear only a limited number of times; long lines must be folded properly. Note that an individual line may have a CRLF sequence inside it because of folding.*

## mail.createMessage( )

Mail **mail.createMessage**( String *from*, String | Array *recipients*, String *subject*, String | Array *content* )

| Parameter | Type | Description |
|---|---|---|
| from | String | Mail address placed in the "From" field of the message |
| recipients | String, Array | Mail address(es) placed in the "To" field of the message |
| subject | String | Subject of the mail |
| content | String, Array | Contents of the message body |
| **Returns** | Mail | New mail object |

### Description

The **mail.createMessage( )** class method builds and returns a *Mail* object which can be used with the **send( )** method.

Pass in *from* a mail address indicating who originally sent the *Mail*. This address will be visible to the *recipients*.

Pass in *recipients* one or more complete mail addresses indicating the recipient(s) of the *Mail*. All recipients will be sent an original copy of the message. Each recipient of the message will see any other mail addresses the message was delivered to.

Pass in *subject* a text value concisely describing the topic covered in detail by the message body.

Pass in *content* a string or an array of string lines. If you pass an array of strings, the new body is considered as the concatenation of all its strings. In this case, the method replaces all single '\r' (CR) characters by blanks and single '\n' (LF) characters by CRLF sequences.

### Example

To create and send a simple email:

```
var username = 'john.smith'; // enter a valid account here
var password = 'mypwx!2';  // enter a valid password here
var address = 'smtp.4dmail.com';
var port = 465;  // SSL port
var mail = require('waf-mail/mail');
```

```
var message = mail.createMessage("from@4d.com", "to@4d.com", "Test", "Hello World!");
message.send(address , port , true, username, password);
```

## mail.Mail( )

void **mail.Mail**( [Object *contents*] )

| Parameter | Type | Description |
|---|---|---|
| contents | Object | JSON object containing mail fields |

### Description

The **mail.Mail( )** method is the constructor of class objects of the *Mail* type. Once defined, such objects can then be sent as messages using the send() method of the SMTP module.

You can build a *Mail* object by using the following syntax:

```
var mail = require('internet/mail');
var myMail = new mail.Mail()
myMail.Subject = 'test'
myMail.From = 'the.sender@4d.com'
myMail.To = '_somebody@4d.com'
myMail.setContent('This is a test');
```

You can also pass a JSON object containing all the fields in the *contents* parameter. The above example could also be written as follows:

```
var mail = require('internet/mail');
var myMail = new mail.Mail(
    {"Subject":"test",
    "From":"the.sender@4d.com",
    "To":"_somebody@4d.com",
    "Content":"This is a test"});
```

## mail.send( )

Boolean **mail.send**( String *address*, Number *port*, Boolean *isSSL*, String *user*, String *password*, String *from*, String | Array *recipients*, String *subject*, String | Array | Mail *content* )

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address (host name or IP address) |
| port | Number | SMTP server port number |
| isSSL | Boolean | true = use SSL, false = do not use SSL |
| user | String | User login name |
| password | String | User password |
| from | String | Email address placed in the "From" field of the message |
| recipients | String, Array | Email address(es) placed in the "To" field of the message |
| subject | String | Subject of the email |
| content | String, Array, Mail | Contents of the message body |
| **Returns** | Boolean | true if the message was sent successfully, false in case of error |

### Description

The **mail.send( )** class method allows you to build and send a message to an SMTP server in a single call. In the event that you require greater control over your message, or if the message is of a more sophisticated nature, you may want to use the other **Mail ***Experimental**** or **SMTP ***Experimental**** methods.

All parameters are mandatory:

- *address*: Host name or IP address of the SMTP server.
- *port*: TCP port number of the SMTP server.
- *isSSL*: Pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *user*: Authentication user name on the SMTP server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack."
- *password*: The authentication password for the *user* on the SMTP server.
- *from*: Email address indicating who originally sent the message. This address will be visible to the *recipients*.
- *recipients*: One or more complete email addresses indicating the recipient(s) of the message. All recipients will be sent an original copy of the message. Each recipient of the message will see any other email addresses the message was delivered to.
- *subject*: Text value concisely describing the topic covered in detail by the message body.
- *content*: A string or an array of string lines, or a *Mail* object with its header and body fields filled.

This function is executed synchronously. It returns **true** if it was completed successfully and **false** if an error occurred.

### Example

With the **mail.send( )** method you can send an email in this simple way:

```
var mail = require('waf-mail/mail');
var rec = ['mark@4d.com' , 'jim@4d.com'];
mail.send('smtp.gmail.com', 465, true, 'joe', 'sdf!f2', 'test@gmail.com', rec, 'test', 'This is my first e
```

## parse( )

void **parse** ( lines , startLine , endLine )

| Parameter | Type | Description |
|---|---|---|
| lines | Array | Lines to parse |
| startLine | Number | Array element index where to start parsing |
| endLine | Number | Last array element index to parse |

#### Description

The **parse( )** method parses the *lines* array and sets the *Mail* with the resulting values. This method is useful to parse an email as received from the POP3 or IMAP protocol.

Pass the received string array of lines in the *lines* parameter.

In *startLine* and *endLine*, pass the indexes from where to start and to end reading from the *lines* array (both are included in the range).

The method returns **true** if the *lines* array was parsed successfully; otherwise it returns **false**.

### removeField( )

void **removeField**( String *name* [, String *value*] )

| Parameter | Type | Description |
|---|---|---|
| name | String | Field name to remove |
| value | String | Field value to remove |

#### Description

The **removeField( )** method removes a field definition from the *Mail* .

In *name*, pass the field name to remove. If the field definition is an array, all matching field elements will be removed. If the *name* field does not exist in *Mail*, the method does nothing.

You can also pass the *value* parameter. In this case, the field will be removed only if both the *name* and *value* parameters match the field definition; otherwise the method does. If the field definition is an array, this will only remove the given *value* from it.

### send( )

void **send**( String *address*, Number *port*, Boolean *useSSL*, String *user*, String *password* )

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address |
| port | Number | SMTP server port number |
| useSSL | Boolean | true = use SSL, false = do not use SSL |
| user | String | User login name |
| password | String | User password |

#### Description

The **send( )** method sends the *Mail* to the specified *address* using the SMTP protocol.

In *address*, pass the name or the IP address of the SMTP server where the message should be sent and, in *port*, pass the TCP port of the server. The *Mail* will be sent to address:port.

In *user*, pass the authentication user name on the SMTP server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack."

In *password*, pass the authentication password for the *user* on the SMTP server.

#### Example

This example sends an email for testing purposes:

```
var username = 'myaccount'; // enter a valid account here
var password = 'mypwx!2';  // enter a valid password here
var address = 'smtp.gmail.com';
var port = 465;  // SSL port for gmail
var mail = require('waf-mail/mail');
var message = new mail.Mail();
message.addField('From', username + '@gmail.com');
message.addField('To', username + '@gmail.com');
message.addField('Subject', 'test');
message.setBody('Hello world!');
message.send(address , port , true, username, password);
```

### setBody( )

void **setBody**( String | Array *body* )

| Parameter | Type | Description |
|---|---|---|
| body | String, Array | New body of the message |

#### Description

The **setBody( )** method sets the *body* part of the *Mail*. The body is made of one or more lines terminated by CRLF sequences. CR and LF characters cannot appear alone inside a body; they must always be in a CRLF sequence.

In *body*, pass a string or an array of string lines (without CRLF at end), correctly formatted according to section 2.3 of the RFC5322 specification.

*Note: Wakanda does not check the body formatting.*

You can also use the **setContent( )** method to generate formatted body content automatically, but **setBody( )** executes faster.

## setContent( )

Boolean **setContent**( String | Array *content* [, Number *lineLimit*] )

| Parameter | Type | Description |
|---|---|---|
| content | String, Array | Contents of the message body |
| lineLimit | Number | Maximum line length |
| **Returns** | Boolean | true if the body contents was correctly set, false otherwise |

### Description

The **setContent( )** method formats and sets the body *content* of the *Mail*.

The *content* of the body can be a single string or an array of string lines. If you pass an array of strings, the new body is considered as the concatenation of all its strings. In this case, the method replaces all single '\r' (CR) characters by blanks and single '\n' (LF) characters by CRLF sequences.

The method checks that each line of the *content* does not exceed the maximum line length. You can set this maximum value using the *lineLimit* parameter. If you omit this parameter, the default maximum length is 998, as specified in the RFC5322 specification.
If a line of the *content* is too long, the current body is left untouched and the **setContent( )** method returns false.

The method returns **true** if it was executed successfully.

Unlike the **setBody( )** method, this method formats and checks the body contents automatically. On the other hand, **setContent( )** executes more slowly. It is usually faster to generate a properly formatted body and use **setBody( )**.

# Net

The Net module provides you with an asynchronous network wrapper, containing methods for handling TCP client sockets. You can include this module with the following statement:

```
net = require("net");
```

Using a TCP socket is based on the following steps:

1. Create a new socket.
2. Connect the socket to a peer or server.
3. Read/write data on the socket.

Reading is asynchronous. When data is available, a 'data' event (see **Events** class) is triggered. The data read is passed to the callback method as an argument. You can use the **addListener( )** method from the **Events** class to set up a callback for the 'data' event.

Writing is synchronous. You just need to use the **write( )** function. Note that written data may still be buffered internally by the operating system.

## remoteAddress

### Description

The **remoteAddress** property returns the remote TCP address of the *socket* as a string.

For example, the following value can be returned:

```
"192.168.93.93"
```

## remotePort

### Description

The **remotePort** property returns the remote port value of the *socket*.

For example, the following numeric value can be returned:

```
8080
```

## bufferSize

### Description

The **bufferSize** property returns the current number of characters in the internal buffer for the *socket*.

Buffered characters are data that is waiting to be written to the *socket*. This automatic internal mechanism allows the socket to be independent from the network connection speed. If the **bufferSize** is getting large, it is a good idea to control its size using the **pause( )** and **resume( )** methods.

## address( )

Object **address**( )

| **Returns** | Object | Bound address and port |
|---|---|---|

### Description

The **address( )** method returns an object containing two attributes, *address* and *port*, representing the address where the *socket* is connected.

For example, the following object could be returned:

```
{"address":"192.168.93.93", "port":8080}
```

## connect( )

void **connect**( Number *port* [, String *host*] [, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| port | Number | TCP port number |
| host | String | Host to connect |
| callback | Function | Callback function to trigger when the connection is established |

### Description

The **connect( )** method opens the connection for the existing *socket* to which it is applied.

In *port*, pass the IP port number to connect to and in *host* you pass the peer machine address. If *host* is omitted, a localhost connection is opened.

*callback* is an optional callback function to trigger when a connection is established. This parameter will be added as a listener for the 'connect' event. Using this parameter is equivalent to writing the following instruction:

```
mySocket.addListener('connect', callback); //sockets implement EventEmitter methods
```

When the connection is established, a 'connect' event is generated. If the connection fails, an 'error' event is generated instead.

#### Example

See example for the **net.Socket( )** constructor function.

## destroy( )

void **destroy**( )

#### Description

The **destroy( )** method closes the *socket* to which it is applied.

After a socket is closed, no data can be sent or received through the socket.

## end( )

void **end**( )

#### Description

The **end( )** method closes the *socket* to which it is applied. This method does the same thing as **destroy( )**.

After a socket is closed, no data can be sent or received through it.

## net.createConnection( )

Socket **net.createConnection**( Number *port* [, String *host*] [, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| port | Number | TCP port number |
| host | String | Host to connect |
| callback | Function | Callback function to trigger when the connection is established |
| **Returns** | Socket | New TCP socket |

#### Description

The **net.createConnection( )** method creates a new TCP connection to *port* on *host*.

In *port*, pass the IP port number to connect to and in *host* you pass the peer machine address. If *host* is omitted, a localhost connection is created.

*callback* is an optional callback function to trigger when a connection is established. This parameter will be added as a listener for the 'connect' event. Using this parameter is equivalent to writing the following instruction:

```
mySocket.addListener('connect', callback); //sockets implement EventEmitter methods
```

The callback function receives one argument: data, which is a *Buffer* object. Buffer objects can handle binary data. You can get a string using its **toString()** function, see class for more details.

When the connection is established, a 'connect' event is generated. If the connection fails, an 'error' event is generated instead.

*Note: You can also create and then connect a socket using two separate statements: net.Socket( ) and connect( ).*

## net.Socket( )

void **net.Socket**( )

#### Description

The **net.Socket( )** method is the constructor of class objects of the *Socket* type. It allows you to create new unconnected **Net** objects on the server, used for establishing client TCP connections.

To create a new socket, just use the following instruction:

```
var mySocket = new net.Socket();
```

Once created, a socket needs to be connected. For this, you have to use the **connect( )** method.

*Note: You can both create and connect a socket at the same time using the net.createConnection( ) method.*

#### Example

In this basic example, we use a socket to read an HTML page through a proxy server:

```
var net = require('net');
// Create a client socket
var socket = new net.Socket();
// Connect the socket
socket.connect(80, 'proxy.private.4d.fr', function () {
```

```
    // We are now connected, we send an HTTP request
    socket.write('GET http://www.google.com/index.html HTTP/1.0\r\n\r\n');
    // The network functions are asynchronous, so we need to set up a callback to read
    // the returned data. Note that even if we set a listener after sending the request,
    // the returned page won't be lost because incoming data is buffered.
    socket.addListener('data', function (data) {
        // Dump the HTML page that was just read
        console.log(data.toString());
        // We are done, request exit from wait()
        exitWait();
    });
});
// Asynchronous execution, wait() is mandatory.
wait();
// Close the socket
socket.destroy();
```

## pause( )

void **pause**( )

### Description

The **pause( )** method pauses the 'data' event triggered for the given *socket*.

The socket is paused until a **resume( )** method is executed.

While paused, data received in the socket is buffered and therefore not lost. However, if a large amount of data is sent to the socket, the pause should not be too long because buffering is not limited.

## resume( )

void **resume**( )

### Description

The **resume( )** method resumes a paused *socket*.

Sockets can be paused using the **pause( )** method.

## setEncoding( )

void **setEncoding**( String *encoding* )

| Parameter | Type | Description |
|---|---|---|
| encoding | String | Encoding for received data |

### Description

The **setEncoding( )** method sets the *encoding* for data received from the *socket* to which it is applied.

*encoding* accepts the following values:

- "ascii"
- "utf8"
- "base64"

## setNoDelay( )

void **setNoDelay**( Boolean *noDelay* )

| Parameter | Type | Description |
|---|---|---|
| noDelay | Boolean | true = disable Nagle's algorithm |

### Description

The **setNoDelay( )** method disables Nagle's algorithm for the *socket* to which it is applied.

By default, sockets use Nagle's algorithm. Since this algorithm works by combining a number of small outgoing messages and sending them all at once, data is buffered before being sent.

Pass *true* to *noDelay* to disable Nagle's algorithm and therefore send data immediately each time **write( )** is called.

## write( )

Boolean **write**( Buffer | String *data* [, String *encoding*] )

| Parameter | Type | Description |
|---|---|---|
| data | Buffer, String | Data to send in the socket |
| encoding | String | Encoding for string data |

| | | |
|---|---|---|
| **Returns** | Boolean | True if data are entirely written to the buffer; False is some data are queued in local memory |

**Description**

The **write( )** method writes *data* to the *socket* to which it is applied. Writing in a socket is a synchronous operation.

The *data* parameter can be one of two types: *Buffer* or *String*. If it is a *String*, the optional *encoding* argument specifies the encoding to use when converting data that is to be sent from JavaScript (string) to binary format. If omitted, the default encoding is UTF8.

This method returns **True** when all the *data* is written successfully to the internal buffer. If only part of the data has been written to the buffer, the method returns **False**. In this case, a 'drain' event is triggered when the buffer is once again free.

**Example**

See example for the **net.Socket( )** constructor function.

# POP3 ***Experimental***

The **POP3** module enables your Wakanda application to retrieve messages from a POP3 email server. Wakanda POP3 methods are MIME compliant and can recognize and extract messages containing multiple enclosures.

To invoke the "POP3" module, you just need to execute the following statement:

```
var pop3 = require("waf-mail/POP3");
```

There are two ways to use this library:

- The short way involves all-in-one methods, **pop3.getAllMail( )** and **pop3.getAllMailAndDelete( )**, which will retrieve all available emails on a POP3 server.
- Or, you can create an empty POP3 object, connect it to a POP3 server and use various individual methods. You may find out the number of message(s) available to be read, their individual sizes, etc.

**Using a Lower Level POP3 Library**
If you need even more control (and are familiar with POP3 protocol), use the **POP3 Client** low-level SSJS module (you will find the **pop3Client.js** module file in the "Modules/waf-mail" Wakanda Server folder). To invoke the "POP3 Client" module, you just need to execute the following statement:

```
var lowpop3 = require("waf-mail/pop3Client");
```

## authenticate( )

void **authenticate**( String *user* , String *password* [, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| user | String | User login name |
| password | String | User password |
| callback | Function | Callback function |

### Description

The **authenticate( )** method allows authenticating the *POP3* object connection on the POP3 server.

In *user*, pass the authentication user name on the POP3 server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack".

In *password*, pass the authentication password for the *user* on the POP3 server.

In *callback*, pass a callback function to be executed when the authentication is done. This function will receive two arguments:

- a Boolean stating whether the authentication has been completed successfully
- an array of line(s) containing the actual reply from the POP3 server. If authentication failed, the reply contains the first failing command (either USER or PASS).

## clearDeletionMarks( )

void **clearDeletionMarks**( Function *callback* )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

### Description

The **clearDeletionMarks( )** method undeletes any message marked as deleted during the current *POP3* connection.

If the server replies successfully, the *callback* function is called and receives two arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server.

## connect( )

void **connect**( String *address*, Number | Undefined *port*, Boolean *isSSL*[, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| address | String | POP3 server address |
| port | Number, Undefined | POP3 server port number (default is 110 if Undefined) |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| callback | Function | Callback function |

### Description

The **connect( )** method connects the *POP3* object to a POP3 email server.

Use the parameters to designate the POP3 server to connect to:

- *address*: Host name or IP address of the POP3 server to connect to.
- *port*: TCP port number of the POP3 server. If this parameter in "undefined", the default value (110) is used.
- *isSSL*: pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *callback*: callback function to be executed when the connection is established. This function will receive two arguments:
  - a Boolean stating whether the server responded (true for OK)
  - an array of line(s) containing the actual reply of the POP3 server.

## createClient( )

POP3 **createClient**( [String *address* [,Number | Undefined *port* [,Boolean *isSSL* [,Function *callback*]]]] )

| Parameter | Type | Description |
|---|---|---|
| address | String | POP3 server address |
| port | Number, Undefined | POP3 server port number (default is 110 for "undefined") |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| callback | Function | Callback function |
| **Returns** | POP3 | New POP3 client object |

### Description

The **createClient( )** method returns a new *POP3* client object.

If you do not pass the optional arguments, the method will return a blank POP3 object, ready to be connected to a POP3 server using the method.

If you pass the optional arguments, the method will return a POP3 object and connect it to the designated POP3 server:

- *address*: Host name or IP address of the POP3 server to connect to.
- *port*: TCP port number of the POP3 server. If this parameter is "undefined", the default value (110) is used.
- *isSSL*: pass true to use SSL to establish the connection with the server, otherwise pass false (default).
- *callback*: callback function to be executed when the connection is established. This function will receive two arguments:
  - a Boolean stating whether the server responded (true for OK)
  - an array of line(s) containing the actual reply of the POP3 server.

## getAllMessageSizes( )

void **getAllMessageSizes**( Function *callback* )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

### Description

The **getAllMessageSizes( )** method allows you to get the size of all messages currently in the mailbox of the open connection referenced in the *POP3* object.

If the server replies successfully, the *callback* function is called and receives three arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server
- the total size of all messages in the mailbox (in bytes)

## getMessageSize( )

void **getMessageSize**( Number *messageNumber*, Function *callback* )

| Parameter | Type | Description |
|---|---|---|
| messageNumber | Number | Number of message whose size you want to get |
| callback | Function | Callback function |

### Description

The **getMessageSize( )** method allows you to get the size of the email designated by *messageNumber*.

Keep in mind that POP3 message numbers start at 1, not zero.

If the server replies successfully, the *callback* function is called and receives three arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server
- the size of the *messageNumber* message (in bytes)

## getStatus( )

void **getStatus**( Function *callback* )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

### Description

The **getStatus( )** method allows you to get the current status of the POP3 server referenced in the *POP3* object.

If the server replies successfully, the *callback* function is called and receives four arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server
- the number of messages available to be read
- the total size of the mailbox (in bytes)

## markForDeletion( )

void **markForDeletion** ( messageNumber , callback )

| Parameter | Type | Description |
|---|---|---|
| messageNumber | Number | Number of the message to delete |
| callback | Function | Callback function |

### Description

The **markForDeletion( )** method allows you to mark the email designated by *messageNumber* to be deleted from the mailbox referenced in the *POP3* object.

The message is not actually deleted until you successfully issue the method (the POP3 client must disconnect from the server for it to do the actual update). If your current connection terminates for any reason (timeout, network failure, etc.) prior to calling the method, any messages marked for deletion will remain on the POP3 server.

In *messageNumber*, pass the number of the message to delete. Keep in mind that message numbers starts at 1, not zero.

If the server replies successfully, the *callback* function is called and receives two arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server.

## pop3.getAllMail( )

Boolean **pop3.getAllMail**( String *address*, Number | Undefined *port*, Boolean *isSSL*, String *user*, String *password*, Array *allMails* )

| Parameter | Type | Description |
|---|---|---|
| address | String | POP3 server address |
| port | Number, Undefined | POP3 server port number (default is 110 if Undefined) |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| user | String | User login name |
| password | String | User password |
| allMails | Array | Empty array (will be filled by the retrieved message(s)) |
| **Returns** | Boolean | true if message(s) read successfully, false in case of error |

### Description

The **pop3.getAllMail( )** class method allows you to connect to a POP3 server and retrieve all available messages in a single call. Unlike the **pop3.getAllMailAndDelete( )** method, after its execution all messages are left untouched on the server (not deleted).

All arguments are required:

- *address*: Host name or IP address of the POP3 server to connect to.
- *port*: TCP port number of the POP3 server. If this parameter is "undefined", the default value (110) is used.
- *isSSL*: pass true to use SSL to establish the connection with the server, otherwise pass false (default).
- *user*: pass the authentication user name on the POP3 server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack".
- *password*: the authentication password for *user* on the POP3 server.
- *allMails*: pass an empty Array in this parameter. It will be filled with the retrieved messages. For example, allMails[0] will be the first mail. The format of the retrieved messages is described in the **retrieveMessage( )** function. You can use the **parse( )** method to parse the mails.

The **pop3.getAllMail( )** function returns true if it was executed successfully. Otherwise, the *allMails* parameter contains the message(s) read until error.

This function is executed synchronously if Wakanda is used.

## pop3.getAllMailAndDelete( )

Boolean **pop3.getAllMailAndDelete**( String *address*, Number *port*, Boolean *isSSL*, String *user*, String *password*, Array *allMails* )

| Parameter | Type | Description |
|---|---|---|
| address | String | POP3 server address |
| port | Number | POP3 server port number (default is 110 if Undefined) |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| user | String | User login name |
| password | String | User password |
| allMails | Array | Empty array (will be filled by the retrieved message(s)) |
| **Returns** | Boolean | true if message(s) read successfully, false in case of error |

### Description

The **pop3.getAllMailAndDelete( )** class method allows you to connect to a POP3 server, retrieve all available messages and delete them on the server in a single call. Unlike the method, after its execution all messages are deleted on the server.

All arguments are required:

- *address*: Host name or IP address of the POP3 server to connect to.
- *port*: TCP port number of the POP3 server. If this parameter is "undefined", the default value (110) is used.
- *isSSL*: pass true to use SSL to establish the connection with the server, otherwise pass false (default).
- *user*: pass the authentication user name on the POP3 server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack".

- *password*: the authentication password for *user* on the POP3 server.
- *allMails*: pass an empty Array in this parameter. It will be filled with the retrieved messages. For example, allMails[0] will be the first mail. The format of the retrieved messages is described in the function. You can use the method to parse the mails.

The **pop3.getAllMailAndDelete( )** function returns true if it was executed successfully. Otherwise, the *allMails* parameter contains the message(s) read until error and all messages are left on the server (not deleted).

This function is executed synchronously if Wakanda is used.

## pop3.POP3( )

void **pop3.POP3**( [String *address* [,Number | Undefined *port* [,Boolean *isSSL* [,Function *callback*]]]] )

| Parameter | Type | Description |
|---|---|---|
| address | String | POP3 server address |
| port | Number, Undefined | POP3 server port number (default is 110 for "undefined") |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| callback | Function | Callback function |

### Description

The **pop3.POP3( )** method creates a new POP3 object. POP3 objects are used to connect to POP3 email servers and retreive messages.

If you do not pass the optional arguments, the method will create a blank POP3 object, ready to be connected to a POP3 server using the [#cmd id="107043"/] method.

If you pass the optional arguments, the method will create a POP3 object and connect it to the designated POP3 server:

- *address*: Host name or IP address of the POP3 server to connect to.
- *port*: TCP port number of the POP3 server. If this parameter is "undefined", the default value (110) is used.
- *isSSL*: pass true to use SSL to establish the connection with the server, otherwise pass false (default).
- *callback*: callback function to be executed when the connection is established. This function will receive two arguments:
  - a Boolean stating whether the server responded (true for OK)
  - an array of line(s) containing the actual reply of the POP3 server.

## quit( )

void **quit**( [Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

### Description

The **quit( )** method issues a QUIT command to log out from the open *POP3* connection.

Logging out from a POP3 server will signal the server that you wish to commit any deletions you made during that session (using the **markForDeletion( )** method). To rollback any deletions you may have made prior to logout, use the **clearDeletionMarks( )** method prior to **quit( )**.

If the server replies successfully, the *callback* function is called and receives two arguments:

- a Boolean stating whether the server responded (true for OK)
- an array of line(s) containing the actual reply of the POP3 server.

## retrieveMessage( )

void **retrieveMessage**( Number *messageNumber*, Function *callback* )

| Parameter | Type | Description |
|---|---|---|
| messageNumber | Number | Number of the message to retreive |
| callback | Function | Callback function |

### Description

The **retrieveMessage( )** method allows you to retrieve the email designated by *messageNumber* from the mailbox of the open connection referenced in the *POP3* object.

Keep in mind that message numbers start at 1, not zero.

If the server replies successfully, the *callback* function is called and receives two arguments. The retrieved mail can be read in the second argument (the reply of the server):

- the first argument is a Boolean stating whether the server responded (true for OK)
- the second argument is an array of line(s) containing the actual reply of the POP3 server. To retrieve the email:
  - ignore the first line which is POP3 protocol specific
  - the content of the email follows with a header (containing fields such as "From", "To", or "Subject"), an empty line separator, followed by the message body.
  - the email is terminated by a dot ('.') on a single line.

You can use the **parse( )** method to parse the retrieved *Mail*.

### Example

This basic example connects to a POP3 server and retrieves the last readable mail:

```
var username = 'myaccount'; // enter a valid account here
var password = 'mypwx!2'; // enter a valid password here
var pop3 = require('internet/POP3');
var toread = new pop3.POP3(); // create a new POP3 object

    // We need to connect to the POP3 server
toread.connect('pop.gmail.com', 995, true, function (isOk, response) { // 995 is the SSL port for gmail
    if (!isOk) {
        close();
        return;
    }

        // Authentication is required first
    toread.authenticate(username, password, function (isOk, response) {
        if (!isOk) {
            close();
            return;
        }

            // We want the last available email
        toread.getStatus(function(isOk, numberMessages, totalSize) {
            if (!isOk) {
                close();
                return;
            }
            console.log(numberMessages + ' messages, taille totale de ' + totalSize + ' octets.');

            toread.retrieveMessage(numberMessages, function (isOk, lines) {
                if (isOk) {
                    var i;
                    for (i = 0; i < lines.length; i++)
                        console.log(lines[i] + '\n');
                }
                close();
            });
        });
    });
});
wait(10000);
toread.quit();
```

## SMTP ***Experimental***

The **SMTP** module allows you to send emails directly through SMTP objects, or by sending messages prepared by the **Mail ***Experimental***** module methods.

To invoke the "SMTP" module, you just need to execute the following statement:

```
var smtp = require("waf-mail/SMTP");
```

There are two ways to use this library:

- The short way involves an all-in-one method, **smtp.send( )**, which will do everything necessary to connect and submit a message to an SMTP server.
- Or, you can create an empty SMTP object, fill it with data and connect it to an SMTP server. This is the more versatile way. In particular, it will allow you to send several emails, and to disconnect when done.

All callbacks receive the same first two arguments: a Boolean telling whether the operation was successful followed by an array of line(s) containing the actual reply by the SMTP server. Some callbacks have additional arguments.

### authenticate( )

void **authenticate**( String *user* , String *password* [, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| user | String | User login name |
| password | String | User password |
| callback | Function | Callback function |

#### Description

The **authenticate( )** method allows authenticating the *SMTP* object connection on the SMTP server. Authentication is required by some SMTP servers in order to reduce the risk that messages have been falsified or that the sender's identity has been usurped, in particular for the purpose of spamming.

*Note: Currently, only LOGIN authentication mode is supported by Wakanda.*

In *user*, pass the authentication user name on the SMTP server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack."

In *password*, pass the authentication password for *user* on the SMTP server.

In *callback*, pass a callback function to be executed when authentication is done. This function will receive two arguments:

- a Boolean stating whether the authentication has been done successfully
- an array of line(s) containing the actual reply of the SMTP server.

### connect( )

void **connect**( String *address*, Number *port*, Boolean *isSSL*, String *domain*[, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address |
| port | Number | SMTP server port number |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| domain | String | Domain name if required |
| callback | Function | Callback function |

#### Description

The **connect( )** method connects the *SMTP* object to an SMTP mail server.

Use the parameters to designate the SMTP server to connect to:

- *address*: Host name or IP address of the SMTP server.
- *port*: TCP port number of the SMTP server.
- *isSSL*: pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *domain*: domain name is required for some SMTP servers. Pass an empty string if it is not needed.
- *callback*: callback function to be executed when the connection is established. This function will receive three arguments:
  - a Boolean stating whether the connection has been done successfully
  - an array of line(s) containing the actual reply of the SMTP server
  - a Boolean stating whether the connected server is ESMTP (true for ESMTP)

*Note: The connect( ) method will always try to use the EHLO command before HELO.*

### quit( )

void **quit**( [Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

#### Description

The **quit( )** method disconnects the *SMTP* object from the SMTP server to which it has been logged.

This method calls the QUIT command internally for the server.

In *callback*, pass a callback function to be executed when the operation is done. This function will receive two arguments:

- a Boolean stating whether the connection has been closed successfully
- an array of line(s) containing the actual reply of the SMTP server.

## send( )

void **send**( String *from*, String | Array *recipients*, Mail *email*[, Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| from | String | Email address of the sender |
| recipients | String, Array | Email address(es) placed in the "To" field of the message |
| email | Mail | Fully completed email object to send |
| callback | Function | Callback function |

### Description

The **send( )** method sends the *email* through the SMTP server to which the *SMTP* object is connected.

In *from*, pass an email address indicating who originally sent the *email*. This address will be visible to the recipients of the *email*.

In *recipients*, pass one or more complete email addresses indicating the recipient(s) of the *email*. All recipients will be sent an original copy of the message. Each recipient of the message will see any other email addresses the message was delivered to.

In *email*, pass a valid *Mail* object prepared using the methods of the **fileName** module. All the fields of the *email* must have been set.

In *callback*, pass a callback function to be executed when the *email* has been sent. This function will receive two arguments:

- a Boolean stating whether the *email* has been sent successfully
- an array of line(s) containing the actual reply of the SMTP server.

## smtp.createClient( )

SMTP **smtp.createClient**( [String *address* [, Number *port*[, Boolean *isSSL*[, String *domain*[, Function *callback*]]]]])

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address |
| port | Number | SMTP server port number |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| domain | String | Domain name if required |
| callback | Function | Callback function |
| **Returns** | SMTP | New SMTP client object |

### Description

The **smtp.createClient( )** class method returns a new *SMTP* client object.

If you do not pass the optional arguments, the method will return a blank SMTP object, ready to be connected to an SMTP server using the **connect( )** method.

If you pass the optional arguments, the method will return an SMTP object and connect it to the designated SMTP server:

- *address*: Host name or IP address of the SMTP server to connect to.
- *port*: TCP port number of the SMTP server.
- *isSSL*: pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *domain*: domain name is required for some SMTP servers. Pass an empty string if it is not needed.
- *callback*: callback function to be executed when the connection is established. This function will receive two arguments:
  - a Boolean stating whether the connection has been done successfully
  - an array of line(s) containing the actual reply of the SMTP server.

### Example

To create a new client SMTP:

```
var smtp = require('waf-mail/SMTP');
var client = smtp.createClient("smtp.gmail.com", 465, true);
```

## smtp.send( )

Boolean **smtp.send**( String *address*, Number *port*, Boolean *isSSL*, String *user*, String *password*, String *from*, String | Array *recipients*, String *subject*, String | Array | Mail *content* )

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address (host name or IP address) |
| port | Number | SMTP server port number |
| isSSL | Boolean | true = use SSL, false = do not use SSL |
| user | String | User login name |
| password | String | User password |
| from | String | Email address placed in the "From" field of the message |
| recipients | String, Array | Email address(es) placed in the "To" field of the message |
| subject | String | Subject of the email |
| content | String, Array, Mail | Contents of the message body |
| **Returns** | Boolean | true if the message was sent successfully, false in case of error |

#### Description

The **smtp.send( )** class method allows you to build and send a message to an SMTP server in a single call. In the event that you require greater control over your message, or if the message is of a more sophisticated nature, you may want to use the other **Mail ***Experimental**** or **SMTP ***Experimental**** methods.

All parameters are mandatory:

- *address*: Host name or IP address of the SMTP server.
- *port*: TCP port number of the SMTP server.
- *isSSL*: Pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *user*: Authentication user name on the SMTP server. *user* should not contain the domain. For example, for the address "jack@4d.com," *user* would just be "jack."
- *password*: The authentication password for the *user* on the SMTP server.
- *from*: Email address indicating who originally sent the message. This address will be visible to the *recipients*.
- *recipients*: One or more complete email addresses indicating the recipient(s) of the message. All recipients will be sent an original copy of the message. Each recipient of the message will see any other email addresses the message was delivered to.
- *subject*: Text value concisely describing the topic covered in detail by the message body.
- *content*: A string or an array of string lines, or a *Mail* object with its header and body fields filled.

This function is executed synchronously. It returns **true** if it was completed successfully and **false** if an error occurred.

### smtp.SMTP( )

void **smtp.SMTP**( [String *address* [, Number *port*[, Boolean *isSSL*[, String *domain*[, Function *callback*]]]]])

| Parameter | Type | Description |
|---|---|---|
| address | String | SMTP server address |
| port | Number | SMTP server port number |
| isSSL | Boolean | true = use SSL for connection, false = do not use SSL |
| domain | String | Domain name if required |
| callback | Function | Callback function |

#### Description

The **smtp.SMTP( )** constructor method creates a new SMTP object. SMTP objects are used to connect to SMTP mail servers and send messages.

If you do not pass the optional arguments, the method will create a blank SMTP object, ready to be connected to an SMTP server using the **connect( )** method.

If you pass the optional arguments, the method will create an SMTP object and connect it to the designated SMTP server:

- *address*: Host name or IP address of the SMTP server to connect to.
- *port*: TCP port number of the SMTP server.
- *isSSL*: pass true to use SSL to establish the connection with the server; otherwise pass false (default).
- *domain*: domain name is required for some SMTP servers. Pass an empty string if it is not needed.
- *callback*: callback function to be executed when the connection is established. This function will receive two arguments:
  - a Boolean stating whether the connection has been done successfully
  - an array of line(s) containing the actual reply of the SMTP server.

### starttls( )

void **starttls**( [Function *callback*] )

| Parameter | Type | Description |
|---|---|---|
| callback | Function | Callback function |

#### Description

The **starttls( )** method upgrades the connection mode of the *SMTP* object to a secured connection.

This methods actually issues the STARTTLS low-level SMTP command. STARTTLS is an extension to communication protocols, which offers a way to upgrade a plain text connection to an encrypted (TLS or SSL) connection instead of using a separate port for encrypted connection. For more information on the STARTTLS command with SMTP, refer to the RFC3207.

In *callback*, pass a callback function to be executed when the upgrade is done. This function will receive two arguments:

- a Boolean stating whether the connection upgrade has been done successfully
- an array of line(s) containing the actual reply of the SMTP server.

Note that the RFC states that the EHLO command should be resent.

# Workers

Wakanda proposes a CommonJS module for creating workers in addition to the standard **Workers** API.

As proposed by CommonJS, the 'worker' module exports **Worker( )** and **SharedWorker( )** properties. These properties are constructors that follow the Web Workers W3C specifications.

To invoke the "worker" module, you just need to execute the following statement:

```
require("worker");
```

For more information about workers management, refer to the **Workers** Wakanda API.

## SharedWorker( )

void **SharedWorker**( String *scriptPath* [, String *workerName*] )

| Parameter | Type | Description |
|---|---|---|
| scriptPath | String | Pathname to JavaScript file |
| workerName | String | Name of the worker to execute |

### Description

The **SharedWorker( )** method is the constructor of the *SharedWorker* type class objects. For more information, please refer to the **SharedWorker( )** section.

## Worker( )

void **Worker**( String *scriptPath* )

| Parameter | Type | Description |
|---|---|---|
| scriptPath | String | Pathname to JavaScript file |

### Description

The **Worker( )** method is the constructor of the dedicated class objects of type *Worker*. For more information, please refer to the **Worker( )** section.