SSJS Modules

Wakanda fully supports the **CommonJS** architecture that allows you to provide additional features through SSJS modules. For more information about the CommonJS architecture, please refer to the CommonJS specification.

Wakanda provides various built-it utility SSJS modules for handling low-level, server-side features, such as TCP connections and events. These modules are described in this section. To access a built-in module and its functions, you just have to call the **require( )** method.

You can also write and use your own SSJS modules, provided that they follow the CommonJS architecture. For more information about using custom SSJS modules, refer to the **Configuring Custom SSJS Modules** section.

Configuring Custom SSJS Modules

Creating and Installing a Module

You can write your own SSJS modules (custom modules). You can also download existing SSJS modules and install them in your Wakanda project. Wakanda SSJS modules must follow the CommonJS modules architecture (for more information about this architecture, please refer to the CommonJS specification):

- a CommonJS module is a JavaScript file (one file per module) containing one or more functions to export.
  Custom Wakanda modules must be stored in a **Modules** folder located at the root of the application (project) folder.
  For example, a "myModule.js" file must be placed at the following location:

  ```
  {project folder}/Modules/myModule.js
  ```

- each public module function must be declared using the **exports** keyword. For example, if you want to provide the following function through a module:

  ```
  function add (a, b) {
       return (a + b);
   }
  ```

  You must include it in the *myModule.js* module file as follows:

  ```
   exports.add = function add (a, b) {
      return (a + b);
   };
  ```

You can add one or more functions in a single module file. You will be able to assign permissions to either the whole module file, or separately to each function (see the following paragraph).

Accessing a Module

Any module functions can be executed using the **require( )** method on the server side. For example, you can write:

```
require('myModule').add()  //SSJS code
  //execute the add function from the myModule CommonJS module
```

Note: If you want your SSJS functions to be available client-side, you have to use JSON-RPC services. For more information, please refer to the Using JSON-RPC Services chapter.

Experimental Mail

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 *****Experimental*** SMTP** 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 (for example if you want to send HTML emails), you may want to use the other *****Experimental*** Mail** or *****Experimental*** SMTP** 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', 'T
```

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);
```

Example

We want to send an email in HTML format:

```
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('Content-Type', 'text/html'); // Specify that body is HTML formatted
message.addField('Subject', 'This is an HTML mail');
message.setBody('<html><B><I>Hello world!</I></B> (normal text)</HTML>'); // Note the HTM
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( )**.

Experimental POP3

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 **connect( )** 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 **quit( )** 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 **quit( )** 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 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 function. You can use the 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
    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 + ' c

            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();
```

Experimental SMTP

The **SMTP** module allows you to send emails directly through SMTP objects, or by sending messages prepared by the *****Experimental*** Mail** 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 (for example if you want to send HTML emails), you may want to use the other *****Experimental*** Mail** or *****Experimental*** SMTP** 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.

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:

```
var events = require("events");
```

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 **net.Socket( )** 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 **addListener( )** 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.

Net - Module

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

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

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
```

net.connect( )

Socket **net.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
Returns	Socket	New TCP socket

Description

The **net.connect( )** method creates a new asynchronous 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.connectSync( )

SocketSync **net.connectSync**(Number *port* [, String *host*])

Parameter	Type	Description
port	Number	TCP port number
host	String	Host to connect
Returns	SocketSync	New synchronous TCP socket

Description

The **net.connectSync( )** method creates a new synchronous 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.

When the connection is established, the function returns a *SocketSync* object that you can handle through the **Net - SocketSync Instances** methods.

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

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 asynchronous TCP socket

Description

The **net.createConnection( )** method creates a new asynchronous 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: and .

net.createConnectionSync( )

SocketSync **net.createConnectionSync**(Number *port* [, String *host*])

Parameter	Type	Description
port	Number	TCP port number
host	String	Host to connect
Returns	SocketSync	New synchronous TCP socket

Description

The **net.createConnectionSync( )** method creates a new synchronous 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.

When the connection is established, the function returns a *SocketSync* object that you can handle through the **Net - SocketSync Instances** methods.

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

net.createServer( )

Server **net.createServer**([Object *options*] [, Function *connectionListener*])

Parameter	Type	Description
options	Object	Allow half open option
connectionListener	Function	Listener for the connection event
Returns	Server	New TCP server

Description

The **net.createServer( )** method creates a new asynchronous TCP server instance. An object of the **Net - Server Instances** type is a *Socket* object that can listen for incoming connections.

In *options*, you can pass an object containing a boolean value for the "allowHalfOpen" property. If you pass **true** for this property, the opened socket will not be automatically closed when the other side of the socket sends a FIN packet. The socket state is then non-readable, but still writable. You should call the **end( )** method explicitly to close it on the server side.
By default, if the *options* parameter is omitted, the property is false:

```
{allowHalfOpen: false}
```

In the optional *connectionListener* listener parameter, you can pass a function that will be automatically triggered when a connection is established. This parameter will be added as a listener for the 'connection' event. Using this parameter is equivalent to writing the following instruction:

```
myServer.addListener('connection', connectionListener);
```

The listener function receives one argument, which is the opened *Socket* object.

Example

Here is an example of an echo server listening for asynchronous connections on port 8082:

```
var net = require('net');
var server = net.createServer(function(socket) { //'connection' event listener
    console.log('server connected');
    socket.on('end', function() {
        console.log('server disconnected');
    });
    socket.write('hello world\r\n');
});
server.listen(8082, function() { //'listening' event listener
    console.log('server bound');
});
```

net.createServerSync( )

ServerSync **net.createServerSync**([Object *options*])

Parameter	**Type**	**Description**
options	Object	Allow half open option
Returns	ServerSync	New synchronous TCP server

Description

The **net.createServerSync( )** method creates a new synchronous TCP server instance. An object of the **Net - ServerSync Instances** type is a *Socket* object that can listen for incoming connections.

In *options*, you can pass an object containing a boolean value for the "allowHalfOpen" property. If you pass **true** for this property, the opened socket will not be automatically closed when the other side of the socket sends a FIN packet. The socket state is then non-readable, but still writable. You should call the **end( )** method explicitly to close it on the server side.
By default, if the *options* parameter is omitted, the property is false:

```
{allowHalfOpen: false}
```

net.isIP( )

Number **net.isIP**(String *address*)

Parameter	**Type**	**Description**
address	String	IP address to test
Returns	Number	4 if address is a valid IP4 address, or 0 if address is invalid

Description

The **net.isIP( )** method allows you to test if *address* is a valid IP address.

The method can return the following values:

- 0 if *address* is an invalid address,
- 4 if *address* is a valid IPv4 address.

Note: IPv6 addresses are currently not supported.

net.isIPv4( )

Boolean **net.isIPv4**(String *address*)

Parameter	Type	Description
address	String	IP address to test
Returns	Boolean	True if address is a valid IPv4 address, false otherwise

Description

The **net.isIPv4( )** method returns **true** if *address* is a valid IP address and **false** otherwise.

net.Server( )

void **net.Server**( )

Description

The **net.Server( )** method is the constructor of class objects of the *Server* type. It allows you to create new unconnected asynchronous **Net - Server Instances** objects, used for accepting client TCP connections.

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

```
var myServer = new net.Server();
```

Once created, a *Server* should listen on a TCP port and address. For this, you have to use the **listen( )** method.

net.ServerSync( )

void **net.ServerSync**( )

Description

The **net.ServerSync( )** method is the constructor of class objects of the *ServerSync* type. It allows you to create new unconnected synchronous **Net - ServerSync Instances** objects. An object of the **Net - ServerSync Instances** type is a *SocketSync* object that can listen for incoming client TCP connections.

To create a new synchronous server, just use the following instruction:

```
var mySyncServer = new net.ServerSync();
```

Once created, a *ServerSync* should listen on a TCP port and address and start accepting connections. For this, you have to use the **listen( )** and **accept( )** methods.

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 - Socket Instances** 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();
```

net.SocketSync( )

void **net.SocketSync**( )

Description

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

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

```
var mySyncSocket = new net.SocketSync();
```

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

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

Net - Server Instances

This class is used to manage asynchronous connections to a TCP *Server*. A *Server* is a *Socket* object that can listen for new incoming connections.

address( )

Object **address**( )

Returns	Object	Bound address and port

Description

The **address( )** method returns the bound address and port of the *Server* as reported by the operating system.

Information is returned in an object containing two properties, "address" (a string) and "port" (a number), for example:

```
{"address":"127.0.0.1", "port":8080}
```

This method is useful for example when the port was assigned randomly by the OS.

Note: You should not call server.address( ) until the 'listening' event has been emitted.

close( )

void **close**( )

Description

The **close( )** method stops the *Server* from accepting new connections.

This function is executed asynchronously: the *Server* is eventually closed when the 'close' *Event* is emitted.

listen( )

void **listen**(Number *port* [, String *host*] [, Function *listeningListener*])

Parameter	Type	Description
port	Number	TCP port to listen, 0 = random
host	String	IP address
listeningListener	Function	Listener for the 'listening' event

Description

The **listen( )** method starts accepting connections on the specified *port* and (optionally) *host*.

Pass the port number to listen in *port* and optionally, a host IPv4 address in *host*. If you pass 0 in *port*, a random port will be used.

This function is executed asynchronously. When the *Server* has been bound, a 'listening' *Event* is emitted. If you pass the *listeningListener* parameter, it will be added as a listener callback for the 'listening' event. The listener callback receives one argument, which is the opened *Socket* object. Using this parameter is equivalent to writing the following instruction:

```
myServer.addListener('listening', callback); //servers implement EventEmitter methods
```

Net - ServerSync Instances

This class is used to manage synchronous connections to a TCP *ServerSync*. A *ServerSync* is a *SocketSync* object that can listen for new incoming connections.

accept( )

SocketSync **accept**([Number *timeOut*])

Parameter	Type	Description
timeOut	Number	Connection timeout
Returns	SocketSync	Synchronous socket connection

Description

The **accept( )** method starts accepting synchronous connections to the *ServerSync* instance and returns the opened *SocketSync* instance.

The port (and host address) defined for the *ServerSync* instance are used for the connection until the **close( )** method is executed, or the specified *timeOut* is reached. If you omit the *timeOut* parameter, no timeout value is applied.

address( )

Object **address**( )

Returns	Object	Bound host address and port

Description

The **address( )** method returns the bound address and port of the *ServerSync* as reported by the operating system.

Information is returned in an object containing two properties, "address" (a string) and "port" (a number), for example:

```
{"address":"127.0.0.1", "port":8080}
```

This method is useful when the port was assigned randomly by the OS.

close( )

void **close**( )

Description

The **close( )** method stops the *ServerSync* from accepting new connections.

This function is executed synchronously: once it is called, the *SocketSync* used on the server port is destroyed.

listen( )

void **listen**(Number *port* [, String *host*])

Parameter	Type	Description
port	Number	TCP port to listen, 0 = random
host	String	IP address

Description

The **listen( )** method sets the *ServerSync* connection to the specified *port* and (optionally) *host*.

Pass the port number to listen in *port* and optionally, a host IPv4 address in *host*. If you pass 0 in *port*, a random port will be used.

Net - Socket Instances

Using a TCP client 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.

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 client *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.

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**.

Example

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

Net - SocketSync Instances

Like asynchronous calls (see **Net - Socket Instances**), synchronous TCP client socket use is based on the following steps:

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

However, unlike for asynchronous calls, both reading and writing are done synchronously. You just need to use the **read( )** and **write( )** 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 client *SocketSync* is connected.

For example, the following object could be returned:

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

connect( )

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

Parameter	**Type**	**Description**
port	Number	TCP port number
host	String	Host to connect

Description

The **connect( )** method opens the connection for the existing *SocketSync* 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.

destroy( )

void **destroy**( )

Description

The **destroy( )** method closes the *SocketSync* method 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 *SocketSync* to which it is applied. This method does the same thing as [#cmd id="105525"/].

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

read( )

Buffer | Null **read**([Number *timeOut*])

Parameter	Type	Description
timeOut	Number	Timeout value in milliseconds
Returns	Buffer, Null	Data read from the socket

Description

The **read( )** method returns in a *Buffer* object the data read from the *SocketSync* instance to which it is applied.

By default, in the *timeOut* parameter was omitted, **read( )** will wait infinitely. You can set a timeout by passing a value to the *timeOut* parameter. The value must be expressed in milliseconds.

The method returns a **Null** value in the following cases:

- the defined timeout was reached,
- the peer has sent FIN,
- an error occured (in this case, the error is thrown).

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 *SocketSync* 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 *SocketSync* 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** (data , encoding)

Parameter	Type	Description
data	Buffer, String	Data to send in the synchronous 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 *SocketSync* to which it is applied.

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**.

Services

Wakanda provides you with a CommonJS utility module that contains helpful functions to manage services. For more information about services, please refer to the **Using Custom Services** chapter.

To access the **services** module functions, you just have to execute:

```
var services = require('services');
```

getInstanceFor( )

Module **getInstanceFor**([Application | Null *application*])

Parameter	Type	Description
application	Application, Null	Target application for the module
Returns	Module	Instance of the target application

Description

The **getInstanceFor( )** method returns an instance of the running module for the application (project) whose name you passed in *application*. This method allows you to use services running in applications other than the current one.

If you omit the *application* parameter or pass **Null**, the method returns an instance of the current module.

getModulePath( )

String **getModulePath**( )

Returns	String	Full path of the module file

Description

The **getModulePath( )** method returns the full path of the CommonJS module file from where it is executed. You can use this utility method in any of your modules, for example to build generic code.

Example

```
var services = require("services");
var mPath = services.getModulePath();
    //mPath contains, for example, 'C:/Wakanda/Wakanda Server/Modules/services'
```

isServiceRegistered( )

Boolean **isServiceRegistered**(String *serviceName*)

Parameter	Type	Description
serviceName	String	Name of the service to check
Returns	Boolean	True if the service is registered; False otherwise

Description

The **isServiceRegistered( )** function returns True if the *serviceName* service is already registered in the running application, and False otherwise.

To register a service, you must either add the appropriate settings in the project settings file (see section **Defining the Settings for a Service**), or using the **registerService( )** function.

postServiceMessage( )

void **postServiceMessage**(String *message*)

Parameter	Type	Description
message	String	Message to post

Description

The **postServiceMessage( )** method allows you to post the *message* to all services running in the current application as well as in other applications of the solution.

registerService( )

Boolean **registerService**(String *serviceName*)

Parameter	Type	Description
serviceName	String	Name of service to register
Returns	Boolean	True if service was registered successfully; False otherwise

Description

The **registerService( )** method registers *serviceName* on-the-fly in the running application.

You can also register a service automatically at startup by declaring it in the settings file of the application (see **Registering a Service** section).

Registration allows a service to communicate with the application through messages and to use specific Storage objects. For more information, please refer to the **Defining the Settings for a Service** section.

TLS - Module

The TLS (*Transport Layer Security*) module provides you with encrypted stream communication capabilities. It is based on OpenSSL.

You can include the TLS module with the following statement:

```
tls = require("tls");
```

tls.connect( )

Socket **tls.connect**(Number *port* [,String *host*] [,Object *options*] [,Function *secureConnectListener*])

Parameter	**Type**	**Description**
port	Number	TCP port number
host	String	Host to connect
options	Object	Connection information
secureConnectListener	Function	Callback listener function for the 'secureConnect' event
Returns	Socket	Secured asynchronous socket

Description

The **tls.connect( )** method creates a new secured client connection on the specified *port* and returns the corresponding *Socket*. For more information, refer to the **Net - Socket Instances** chapter.

You can pass in *host* the peer machine address. If *host* is omitted, a localhost connection is opened.

Pass in *options* an object that includes the following members:

- *key*: string or *buffer* containing the private key of the client in PEM format
- *passphrase*: string of passphrase for the private key
- *cert*: string or *buffer* containing the certificate key of the client in PEM format.
- *ca*: array of strings or *buffer* of trusted certificates. If omitted, several well known "root" certificates will be used, like VeriSign. These are used to authorize connections.
- *NPNProtocols*: array of string or *buffer* containing supported NPN protocols. Buffer should have following format: 0x05hello0x05world, where first byte is next protocol name's length. Passing array should usually be simpler: ['hello', 'world'].
- *servername*: Servername for SNI (Server Name Indication) TLS extension.

secureConnectListener is an optional callback function to trigger after the new connection has been successfully handshaked. This parameter will be added as a listener for the 'secureConnect' event. Using this parameter is equivalent to writing the following instruction:

```
myTlsSocket.addListener('secureConnect', secureConnectListener );
    //sockets implement EventEmitter methods
```

tls.connectSync( )

SocketSync **tls.connectSync**(Number *port* [, String *host*] [, Object *options*])

Parameter	**Type**	**Description**
port	Number	TCP port number
host	String	Host to connect
options	Object	Connection information
Returns	SocketSync	Secured synchronous socket

Description

The **tls.connectSync( )** method creates a new secured client connection on the specified *port* and returns the corresponding *SocketSync*. For more information, refer to the **Net - SocketSync Instances** chapter.

You can pass in *host* the peer machine address. If *host* is omitted, a localhost connection is opened.

Pass in *options* an object that includes the following members:

- *key*: string or *buffer* containing the private key of the client in PEM format
- *passphrase*: string of passphrase for the private key
- *cert*: string or *buffer* containing the certificate key of the client in PEM format.
- *ca*: array of strings or *buffer* of trusted certificates. If omitted, several well known "root" certificates will be used, like VeriSign. These are used to authorize connections.

- *NPNProtocols*: array of string or *buffer* containing supported NPN protocols. Buffer should have following format: 0x05hello0x05world, where first byte is next protocol name's length. Passing array should usually be simpler: ['hello', 'world'].
- *servername*: Servername for SNI (Server Name Indication) TLS extension.

tls.createServer( )

Server **tls.createServer**(Object *options* [, Function *secureConnectionListener*])

Parameter	**Type**	**Description**
options	Object	Server information
secureConnectionListener	Function	Callback listener function for the 'secureConnection' event
Returns	Server	New asynchronous secured server

Description

The **tls.createServer( )** method creates a new secured server and returns the corresponding asynchronous *Server*. For more information, refer to the **Net - Server Instances** chapter.

Pass in *options* an object that can include the following members:

- *key*: string or *Buffer* containing the private key of the server in PEM format. (Required)
- *passphrase*: string of passphrase for the private key.
- *cert*: string or *Buffer* containing the certificate key of the server in PEM format. (Required)
- *ca*: array of strings or *Buffers* of trusted certificates. If omitted, several well known "root" certificates will be used, like VeriSign. These are used to authorize connections.
- *ciphers*: string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format.
- *requestCert*: boolean. If true the server will request a certificate from clients that connect and attempt to verify that certificate. Default: false.
- *rejectUnauthorized*: boolean. If true the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if *requestCert* is true. Default: false.
- *NPNProtocols*: array or *Buffer* of possible NPN protocols. (Protocols should be ordered by their priority).
- *sessionIdContext*: string containing a opaque identifier for session resumption. If *requestCert* is true, the default is MD5 hash value generated from command-line. Otherwise, the default is not provided.

secureConnectionListener is an optional callback function to trigger after the new connection has been successfully handshaked. This parameter will be added as a listener for the 'secureConnection' event. Using this parameter is equivalent to writing the following instruction:

```
myTlsServer.addListener('secureConnection', secureConneciontListener );
    //servers implement EventEmitter methods
```

tls.createServerSync( )

ServerSync **tls.createServerSync**(Object *options*)

Parameter	**Type**	**Description**
options	Object	Server information
Returns	ServerSync	New synchronous secured server

Description

The **tls.createServerSync( )** method creates a new secured synchronous server and returns the corresponding synchronous *ServerSync*. For more information, refer to the **Net - ServerSync Instances** chapter.

Pass in *options* an object that can include the following members:

- *key*: string or *Buffer* containing the private key of the server in PEM format. (Required)
- *passphrase*: string of passphrase for the private key.
- *cert*: string or *Buffer* containing the certificate key of the server in PEM format. (Required)
- *ca*: array of strings or *Buffers* of trusted certificates. If omitted, several well known "root" certificates will be used, like VeriSign. These are used to authorize connections.
- *ciphers*: string describing the ciphers to use or exclude. Consult http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT for details on the format.
- *requestCert*: boolean. If true the server will request a certificate from clients that connect and attempt to verify that certificate. Default: false.
- *rejectUnauthorized*: boolean. If true the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if *requestCert* is true. Default: false.

- *NPNProtocols*: array or *Buffer* of possible NPN protocols. (Protocols should be ordered by their priority).
- *sessionIdContext*: string containing a opaque identifier for session resumption. If *requestCert* is true, the default is MD5 hash value generated from command-line. Otherwise, the default is not provided.

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.