Global Application

Introduction

The **Global application** theme gathers basic classes for managing Wakanda applications that are executed on the server.

Note: A Wakanda "application" is a Wakanda "project" that is being executed.

The global space for Wakanda applications includes various classes, properties and methods used to control and manage different aspects concerning the publication of your Web applications:

- **Application**: main class of the API that provides numerous utility properties and methods for retrieving information about the solution being executed, the services that are enabled and the location of the folders and files used on the server; this class also provides basic functions for processing and converting data, images and texts as well as functions for creating and opening datastores.
- **HTTP Server**: properties and methods that control the operation of the integrated HTTP server of Wakanda Server.
- **Solution**: properties and methods for managing the solution that is open on the server.

Application

The Application class describes each application object interface and provides methods for managing the published application.

administrator

Description

This property is set to *true* if the current application is defined as the solution's administration application.

The solution's administration application is defined through the project's settings file (*projectName*.waSettings).

By default, this property returns *false*. If none of the projects in the solution has been defined as the administration application, Wakanda provides a default interface for administration purposes.

console

Description

The console of the application provides an interface to log JavaScript actions.

This property returns an object of the *Console* class. For more information, refer to the **Console** documentation.

Example

The following example adds two messages to the console: one to display information and the other an error:

```
console.info("Update successful", product.name);
console.error("Update failed", product.name, "(Error:", error, ")");
```

dataService

Description

The **dataService** property gives access to the Data Service for the current application. For more information about the properties and methods that can be used with the *DataService* object, refer to the **DataService** class description.

This service activates the HTTP REST interface to the Wakanda Datastore. It is recommended to start this service so it can be used by the Wakanda Ajax Framework.

ds

Description

The **ds** property is a reference to the application's default datastore. This reference can be used in your server-side code as a shortcut to reference the default datastore.

Example

The following example applies the max function to the default datastore's Employee class:

```
var maxSalary = ds.Employee.max('salary');
//Get maximum salary from the Employee class in the default datastore
```

httpServer

Description

The **httpServer** property gives access to the HTTP Server object for the current application. For more information about the properties and methods that can be used with the HTTP Server object, refer to the **HTTP Server** class description.

Example

The following example obtains the IP Address of the HTTP server:

```
var serverIP = application.httpServer.ipAddress;
```

name

Description

The **name** property returns the name of the current application.

By default, the application name is the name of the *.waProject* file without its extension.

rpcService

Description

The **rpcService** property gives access to the RPC Service for the current application. For more information about the properties and methods that can be used with the RPCService object, refer to the **RPCService** class description.

For more information about JSON-RPC implementation in Wakanda, refer to the **Using JSON-RPC Services** manual.

sessionStorage

Description

The **sessionStorage** property is the *Storage* object available for each HTTP session in the current application.

The **sessionStorage** property gives you an easy way to handle user sessions and to keep session-related data on the server. This data is accessible at the session level (each session has its own **sessionStorage** object).

A new session is opened on the server when you make a call to **sessionStorage** for the first time. A special cookie is then sent to the browser with a reference to the *Storage* object. Data stored in **sessionStorage** is alive while the HTTP session exists.
A session is closed when the cookie expires or when you call the **clear( )** method on the *Storage* object.

Note: You can also access the Storage object that is available for the entire Wakanda project by using the storage property.

Storage objects have specific properties and methods, listed in the **Storage** class description.

solution

Description

The **solution** property is an object containing the current solution published by Wakanda Server. A solution can run one or more applications.

This property allows you to access the applications associated with the current solution. For more information about the **solution** object, refer to the **Solution** class description.

Example

This example returns the number of applications running on the server:

```
var numRunningApplications = 0
solution.applications.forEach(
    function (app) {
        numRunningApplications += (app.httpServer.started) ? 1 : 0;
    }
);
```

webAppService

Description

The **webAppService** property gives access to the Web Application Service for the current application.

For more information about the properties and methods that can be used with the WebAppService object, refer to the **WebAppService** class description.

storage

Description

The **storage** property returns the project *Storage* object for the current application.

Data stored in **storage** is alive while the project (application) is running. This data is shared between all user sessions. You can use locking methods to handle multiple access.

Note: You can also have access to Storage objects that are available for each user session by using the sessionStorage property.

Storage objects have specific properties and methods, listed in the **Storage** class description.

settings

Description

The **settings** property contains the application's current project settings in the form of a *Storage* object. Current settings are shared by all sessions in the project and are persistent until the project is closed.

The **settings** object implements the interface so that you can get pairs of keys/values by using the method. A value is generally an object containing a key/value pair. You can only read the information from the settings; you cannot change them.

Here is the list of keys and values available for the **settings** object. The example of settings values are those created for the blank project template.

Key	Value type	Value properties (for objects)	Value example
project	object	publicName	""
		listen	"0"
		hostName	"localhost"
		responseFormat	"json"
		administrator	"false"
		servers.http.started	"true"
http	object	port	"8082"
		SSLCertificatePath	""
		allowSSL	"false"
		SSLMandatory	"false"
		SSLPort	"443"
		SSLMinVersion	"3"
		useCache	"false"
		pageCacheSize	"5192"

		maximumProcess	"100"
		maximumRequestsByConnection	"100"
		maximumTimeout	"15"
		inactiveProcessTimeout	"5"
		acceptKeepAliveConnections	"true"
		sendExtendedCharacterSetDirectly	"false"
		standardSet	"UTF-8"
		logFormat	"ELF"
		logPath	"Logs/"
		logMaxSize	"10000"
webApp	object	enabled	"true"
		pattern	"/"
		documentRoot	"webFolder/"
		directoryIndex	"index.html"
dataService	object	enabled	"true"
		pattern	"/rest/"
rpcService	object	enabled	"true"
		pattern	"/rpc/"
javaScript	object	reuseContexts	"true"
resources	array	arr[0].location	"/walib/"
		arr[0].lifeTime	"31536000"

For more information about these settings, refer to the **Wakanda Server Administration** manual.

Example

To know the http port of the project settings:

```
var httpObj = settings.getItem( "http" );
var httpPort = httpObj.port
```

os

Description

The **os** property returns an object containing information about the current operating system.
This object contains the following properties:

Property	Type	Value
isMac	Boolean	true or false
isWindows	Boolean	true or false
isLinux	Boolean	true or false

application

Description

The **application** property returns an *Application* global object, describing each Wakanda application object interface.

directory

Description

The **directory** property is a reference to the application's default *Directory*. By default, the application Directory object is built upon the directory file of the solution (named *solutionName*.waDirectory). It is shared by all Wakanda applications of the solution. This reference can be handled in your server-side code with the methods and properties of the **Directory** class.

Example

We want to select all the users whose name starts with "S". We need to do it through the **internalStore** property of the Directory object, which contains the users and groups:

```
var pusers = directory.internalStore.User.query( "name = :1" , "S@" );
// User is one of the datastore classes in internalStore
```

process

Description

The **process** property returns an object containing some information about the running version of Wakanda.
This object contains the following read-only properties:

Property	Type	Description
buildNumber	Number	Internal build number, for example 106683
version	String	Wakanda version number, for example "2.0"

addHttpRequestHandler( )

void **addHttpRequestHandler**(String *pattern*, String *filePath*, String *functionName*)

Parameter	Type	Description
pattern	String	Pattern of the request to intercept
filePath	String	Path to the file in which the handler function is defined
functionName	String	Name of the function to handle the request matching the pattern

Description

The **addHttpRequestHandler( )** method installs a request handler function on the server. Once installed, this function will intercept and process any incoming HTTP request matching a predefined pattern.

- In the *pattern* parameter, pass a string describing the HTTP requests that you want to intercept. This pattern should be defined through a JavaScript Regex (Regular expression). For more information, see the following paragraph.
- In the *filePath* parameter, pass a string containing the path to the file that has the function to call for this handler. You can pass either an absolute path or a path relative to the project folder (POSIX syntax).
- In the *functionName* parameter, pass the name of the request handler function to call when it matches the *pattern*. This function will receive two object parameters (*request* and *response*) and can return a value.

For a complete description of the server-side HTTP request handlers feature, refer to the **Introduction to HTTP Request Handlers** section.

Example

If you write in the bootStrap.js file:

```
addHttpRequestHandler('(?i)^/doGetStuff$', 'myFile.js', 'myFunction');
```

... the *myFunction* request handler will be called each time the server receives a query containing the "/doGetStuff" URL.

BinaryStream( )

BinaryStream **BinaryStream**(String | File *file*, Longint *realFormat*)

Parameter	Type	Description
file	String, File	Binary file to reference
realFormat	Longint	Streaming action: "Write" to write data, "Read" to read data
Returns	BinaryStream	New BinaryStream object

Description

The **BinaryStream( )** method creates a new object of the **BinaryStream Class**. For more information, refer to the **BinaryStream( )** method description from the **BinaryStream Class**.

Buffer( )

void **Buffer**(Number | Array | String *definition* [, String *encoding*])

Parameter	Type	Description
definition	Number, Array, String	Size (number or array) of the buffer or string to set to the buffer
encoding	String	Encoding method (if a string is passed in definition)

Description

The **Buffer( )** method is the constructor of the class objects of the *Buffer* type. It allows you to create new *Buffer* objects on the server.

You can create a new *Buffer* with one of the following constructors:

- This constructor creates a new buffer of a certain *number* of bytes. The buffer contents are undefined.

  ```
  var myBuffer = new Buffer( number );
  ```

- This constructor creates a new buffer from an array object. The buffer contents are the elements of the given *byteArray*.

  ```
  var myBuffer = new Buffer( byteArray );
  ```

- This constructor creates a new buffer from a string. The buffer content is the string. In this case, you can provide an *encoding* method for the string. By default, 'utf8' is used.

  ```
  var myBuffer = new Buffer( myString, encoding );
  ```

  Available values for *encoding* are:
  - 'ascii' - for 7-bit ASCII data only. This encoding method is very fast, and will strip the high bit if set.
  - 'utf8' (default value) - Multi-byte encoded Unicode characters.
  - 'ucs2' - 2-bytes, Little Endian encoded Unicode characters. It can encode only BMP (Basic Multilingual Plane, U+0000 - U+FFFF).
  - 'base64' - Base64 string encoding.
  - 'hex' - Encode each byte as two hexadecimal characters.

Example

This example creates a buffer of 16KB filled with 0xFF:

```
var vData =new Buffer(16*1024);
vData.fill(0xFF);
```

clearInterval( )

void **clearInterval**(Number *timerID*)

Parameter	Type	Description
timerID	Number	Scheduler to cancel

Description

The **clearInterval( )** method cancels the *timerID* scheduler previously set by the **setInterval( )** method.

If *timerID* does not correspond to an active scheduler, the method does nothing.

Note: The Wakanda **clearInterval( )** method is compliant with the Timers W3C specification.

clearTimeout( )

void **clearTimeout**(Number *timerID*)

Parameter	Type	Description
timerID	Number	Timeout to cancel

Description

The **clearTimeout( )** method cancels the *timerID* timeout previously set by the **setTimeout( )** method.

If *timerID* does not correspond to an active timeout, the method does nothing.

Note: The Wakanda **clearTimeout( )** method is compliant with the Timers W3C specification.

close( )

void **close**( )

Description

The **close( )** method ends the thread from which it is called.

This method can be called:

- From a *Worker* or a *SharedWorker* parent thread where only the parent thread is closed.
  If you want to close a dedicated child worker from the parent thread, you can call the **terminate( )** method. If you call **close( )** on a waiting parent thread, all the dedicated workers spawned from that thread will receive a message to terminate. If it is called during a callback in a **wait( )**, this will exit it.
- From a child thread. In this case, the thread is closed.

The **close( )** method effect is not immediate: the JavaScript interpreter will continue until finishing the current execution (exiting the current callback). All resources will then be freed up.

createDataStore( )

Datastore **createDataStore**(File *model*, File *data*)

Parameter	Type	Description
model	File	Reference to an existing datastore model file
data	File	Reference of data file to be created
Returns	Datastore	Reference of datastore object that was created

Description

The **createDataStore( )** method creates a new *Datastore* object based on the *model* passed as a parameter and returns a reference to this datastore.

Note: This method only creates a new data file -- not a datastore model file. As described in What is a datastore?, you cannot create or build datastore model files with JavaScript.

- Pass the reference to an existing datastore model file on disk (the file ending with .waModel) in the *model* parameter.
- Pass the reference of the data file (file ending with .waData) to be created on the disk of the server in the *data* parameter.

Note: You can generate file references using the File method. For more information, refer to the description of Files and Folders.

This method creates the data file at the location specified using the *data* parameter and returns a *Datastore* reference to the new datastore. You can assign this reference to a variable that you can then use as the target object for queries, just like you use the **ds** object (see example). If you use a variable named *ds*, it will override the current datastore.

Example

This example creates a new blank datastore on disk using the current datastore model:

```
    // get a reference to the current datastore model file
var currentFolder = ds.getModelFolder().path
```

```
var currentModel = File(currentFolder+ ds.getName() + ".waModel");
            // only works if the datastore has the same name as the model file
if (currentModel.exists)     // if the model actually exists
{
    var dataFile = File(currentFolder+ "newData.waData");     // create a reference to the new data file
    var myDS = createDataStore(currentModel, dataFile);     // create and reference the datastore
}
```

currentSession( )

ConnectionSession **currentSession**( )

Returns	ConnectionSession	Object containing the current user session properties

Description

Note: In Wakanda v1, this method belongs to the Datastore class.

The **currentSession( )** method returns an object of the *ConnectionSession* type identifying the current session under which the current user is actually running on the server. Because of the "promote" mechanism, running session privileges can be temporarily different from standard user privileges, defined in the **Directory** of the application. A different session is created for each thread connected to the server. For more information about the "promote" mechanism in Wakanda, please refer to the **Assigning Group Permissions** section.

Objects of the *ConnectionSession* type can be handled through the methods and properties of the **Session** theme.

If this method is not executed from within the context of a user session, a *null* value is returned.

currentUser( )

User **currentUser**( )

Returns	User	Object containing the current user properties

Description

Note: In Wakanda v1, this method belongs to the Datastore class.

The **currentUser( )** method returns the user who opened the current user session on the server. The returned object includes the name, ID and groups of the user. Objects of the *User* type can be handled through the methods and properties of the **User** class.

If this method is not executed within the context of a user session, an empty string is returned for the user name.

For more information about the User and groups management in Wakanda, please refer to chapter **Users and Groups**.

Example

The following datastore class method, named *getCurrentUser*, can be called to display the full name of a user in an interface:

```
model.Person.methods.getCurrentUser = function()
{
    var result = currentUser();
    return result.fullName;
}
```

dateToIso( )

String **dateToIso**(Date *dateObject*)

Parameter	**Type**	**Description**
dateObject	Date	Date object to convert
Returns	String	ISO string date

Description

The **dateToIso( )** method converts the JavaScript date object you passed in the *dateObject* parameter into an ISO format string. This function is useful when you need to display dates in widgets, for example.

You can convert an ISO date back to a JavaScript object using the **isoToDate( )** function.

Example

```
var isoString = dateToIso(new Date());
```

displayNotification( )

void **displayNotification**(String *message* [, String *title*] [, Boolean *critical*])

Parameter	**Type**	**Description**
message	String	Message to display
title	String	Title of window
critical	Boolean	True for critical information

Description

The **displayNotification( )** method allows you to display a system notification window on the server machine.

Pass the message to display in the *message* parameter and the window title in the *title* parameter ("Notification" is used by default as the title).

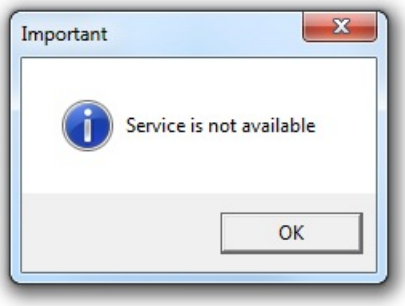
An alert dialog is shown on Mac OS and Windows. On Windows, an "info" icon is shown by default. If you set the *critical* parameter to *true*, a "warning" icon is used.

Example

When you execute the following code:

```
displayNotification("Service is not available" , "Important");
```

...the following alert box is displayed on the server (Windows):



exitWait( )

void **exitWait**( )

Description

The **exitWait( )** method stops all pending **wait( )** loops in the thread from which it is called.

wait( ) loops are necessary to allow asynchronous communication between threads. To exit from such a loop once the callback method has been executed, you need to call the **exitWait( )** method. For an example of asynchronous communication, refer to the **net.Socket( )** method description.

File( )

File **File**(String | Folder *path* [, String *fileName*])

Parameter	Type	Description
path	String, Folder	Path of the file to reference
fileName	String	Name of the file to reference
Returns	File	New File object

Description

The **File( )** method creates a new *File* object. *File* objects are handled using the various properties and methods of the **File Class**.

In the *path* parameter, pass the file's path in one of the following forms:

- an absolute path (using the "/" separator) or a URL, including the file name -- in both cases, the *fileName* parameter must be omitted
- a *Folder* object -- in this case, the *fileName* parameter must be passed

If necessary, pass the file name in the *fileName* parameter.

Note that this method only creates an object that references a file and does not create the file on disk. You can work with *File* objects referencing files that may or may not exist. If you want to create the referenced file, you need to execute the **create( )** method.

Example

This example creates a new blank datastore on disk using the current datastore model:

```
    // get a reference to the current datastore model file
var currentFolder = ds.getModelFolder().path
var currentModel = File(currentFolder+ ds.getName() + ".waModel");
            // only works if the datastore has the same name as the model file
if (currentModel.exists)     // if the model actually exists
{
    var dataFile = File(currentFolder+ "newData.waData");     // create a reference to the new data file
    var myDS = createDataStore(currentModel, dataFile);     // create and reference the datastore
}
```

Folder( )

Folder **Folder**(String *path*)

Parameter	Type	Description
path	String	Path of the folder to reference
Returns	Folder	New Folder object

Description

The **Folder( )** method creates a new object of type *Folder*. *Folder* objects are handled using the various properties and methods of the **Folder Class**.

In the *path* parameter, pass the path of the folder to reference. It can be:

- an absolute path in Posix syntax (using the "/" separator) or
- a URL.

Note that this method only creates an object that references a folder and does not create anything on disk. You can handle *Folder* objects referencing folders that may or may not exist. If you want to create the referenced folder, you need to execute the **create( )** method.

Example

This example creates an "Archives" subfolder in the "Wakanda" folder:

```
var newFolder = Folder ("c:/Wakanda/Archives/");
var isOK = newFolder.create();
```

garbageCollect( )

void **garbageCollect**( )

Description

The **garbageCollect( )** method launches the garbage collector on all sleeping contexts.

generateUUID( )

String **generateUUID**( )

Returns	String	UUID String (32 chars)

Description

The **generateUUID( )** method returns a valid UUID string (32 chars) that you can use for your application needs.

An UUID is a 16-byte number (128 bits). It contains 32 hexadecimal characters. The **generateUUID( )** method returns an UUID expressed as a string containing a combination of 32 letters [A-F, a-f] and numbers [0-9], for example 550e8400e29b41d4a716446655440000 (non-canonical form).

Example

This example generates a UUID:

```
var aString = generateUUID();
    // returns for example "9AE457F4B557BD7895AD4712345ABCDE"
```

getDataStore( )

Datastore **getDataStore**(String *projectName*)

Parameter	Type	Description
projectName	String	Full access path of the project to use
Returns	Datastore	Reference to datastore of projectName

Description

This method returns a reference to the current datastore of the project whose name you passed in *projectName*. This project must belong to the current solution (and thus be opened by the server).

With this method, you can use project B's current datastore from project A if they are both in the same solution. This comes in handy when your solution contains several complimentary projects and you want to be able to access one project's data from another project.

You pass a string of characters indicating the project name in *projectName*.

Example

The following example returns a reference to the datastore in the "Clients" project from the current project:

```
var dsTemp = getDataStore ("Clients");     // get a reference to the datastore of the "Clients" project
var mySet = dsTemp.Clients.query (country = "France");     // perform a search in this datastore
```

Notes:

- You can get the same information by using the *ds* global application property with the **getApplicationByName( )** method (which belongs to the **Solution** class). You may find this alternative useful for example when a project name is changed in the application.

  ```
  var dsTemp = solution.getApplicationByName ("Clients").ds;   // return the datastore in the "Clients" appl
  ```

- Another alternative using the **applications** property in the **Solution** class:

  ```
  var dsTemp = solution.applications[1].ds;   // return the datastore of the second application loaded
  ```

getFolder( )

Folder | String **getFolder**([String *kind* [, Boolean | String *format*]])

Parameter	Type	Description
kind	String	Type in which the folder must be returned: Folder object (if omitted), path or url
format	Boolean, String	True or "posix" or "encoded" (default); False or "system" or "notEncoded"
Returns	Folder, String	Folder containing the application file

Description

The **getFolder( )** method returns the folder containing the application file (i.e., the project file with the .waSettings extension).

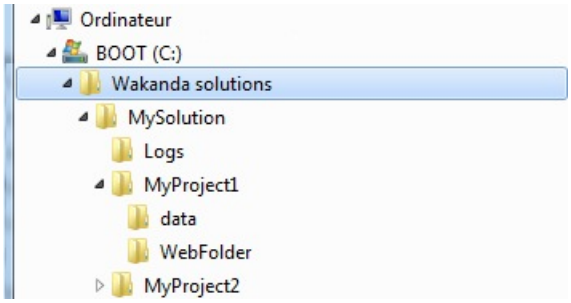
You can get the folder in different formats, depending on the string you pass in the *kind* parameter:

- If you omit the *kind* parameter, the method returns an object of the *Folder* type (see the **Folder Class** methods and properties).
- If you pass "path", the method returns a string containing the path of the folder.
  By default in this case, the path is expressed in posix syntax. You can get this value in system syntax by passing *false* or "system" in the *format* parameter (you can set the standard format back by passing *true* or "posix" in this parameter).
- If you pass "url", the method returns a string containing the URL of the folder.
  By default in this case, the URL is encoded. You can get this value without encoding by passing *false* or "notEncoded" in the *format* parameter (you can back the standard format back by passing *true* or "encoded" in this parameter).

Note: On Mac OS, the posix and system paths are equivalent.

Example

Considering the following organization of files and folders on your disk:



The following example illustrates the different values that can be returned by applying **getFolder( )** to an application:

```
var theEncodedURL=getFolder("url");
    // returns file:///C:/Wakanda%20solutions/MySolution/MyProject1/
var theRawURL=getFolder("url", False);
    // returns file:///C:/Wakanda solutions/MySolution/MyProject1/
var thePath=getFolder("path");
     // returns C:/Wakanda solutions/MySolution/MyProject1/
var theFolder=getFolder( );
     // returns { name: "MyProject1", path: "C:/Wakanda solutions/MySolution/MyProject1/", folders: Array(), 11
```

getItemsWithRole( )

File | Folder | Array **getItemsWithRole**(String *role*)

Parameter	Type	Description
role	String	Role for which you want to get the current item
Returns	Array, File, Folder	Item with the defined role

Description

The **getItemsWithRole( )** method returns the item associated with the *role* you passed as a parameter.

Depending on the *role* you passed, the method can return a *File*, a *Folder*, or an array of *File* objects.

Here are the available strings that you can pass in *role*:

role	Type	Description
"settings"	File	XML file defining the settings for your project (.waSettings)
"catalog"	File	Model file of the application (.waModel)
"data"	File	Application's current data file (.waData)
"rpc"	File or array of Files	JavaScript file(s) containing functions that will be available as JSON-RPC services
"catalogRpc"	File	JSON schema file describing the JSON-RPC service functions
"bootStrap"	File or array of Files	JavaScript file(s) automatically executed at the project launch (can contain HTTP request handlers)
"webFolder"	Folder	Folder containing all the files that Wakanda will send to the Web
"smartphone"	Folder	Folder containing all the files used for smartphone interfaces
"tablet"	Folder	Folder containing all the files used for tablet interfaces

getProgressIndicator( )

ProgressIndicator **getProgressIndicator**(String *name*)

Parameter	Type	Description
name	String	Unique name of the ProgressIndicator object on the server
Returns	ProgressIndicator	Existing ProgressIndicator object on the server

Description

The **getProgressIndicator( )** method returns the *ProgressIndicator* type object whose name you passed in the *name* parameter. The object must have been previously created by using the **ProgressIndicator( )** method.

The **getProgressIndicator( )** method connects a processing task to an existing progressIndicator and is particularly useful for .

getSettingFile( )

File | String **getSettingFile**(String *settingID* [, String *kind*[, Boolean | String *format*]])

Parameter	Type	Description
settingID	String	ID of the setting for which to get the file
kind	String	Type in which the file must be returned: File object (if omitted), path, relativePath or url
format	Boolean, String	True or "posix" or "encoded" (default); False or "system" or "notEncoded"
Returns	File, String	File containing the settings

Description

The **getSettingFile( )** method returns a reference or the path to the file containing the application setting whose ID you passed in *settingID*. For example, if you want to get the .waSettings file containing the data service settings, pass "dataService" in *settingID*.

The available setting IDs are :

- "http": HTTP server settings including port, SSL parameters, etc.
- "dataService": data service status and pattern
- "rpcService": rpc service status and pattern
- "webApp": web application service status and parameters
- "resources": location of the application's resources
- "javaScript": JavaScript context parameter

By default, all the application settings are gathered in a single settings file, named *projectName*.waSettings -- any of the listed setting IDs will return the same information. But they could be stored in separate .waSettings files in the project, that's why you have to define which one in the *settingID* parameter.

You can get the file in different formats, depending on the string you pass in the *kind* parameter:

- If you omit the *kind* parameter, the method returns an object of the *File* type (see the **File Class** methods and properties).
- If you pass "path" or "relativePath", the method returns a string containing the path to the file (absolute or relative to the project folder)).
  By default in this case, the path is expressed in posix syntax. You can get this value in system syntax by passing *false* or "system" in the *format* parameter (you can set the standard format back by passing *true* or "posix" in this parameter).
- If you pass "url", the method returns a string containing the URL of the file.
  By default in this case, the URL is encoded. You can get this value without encoding by passing *false* or "notEncoded" in the *format* parameter (you can set the standard format back by passing *true* or "encoded" in this parameter).

Note: On Mac OS, posix and system paths are equivalent.

Example

This example returns the path of the file containing the rpc service settings:

```
var rpcsets = getSettingFile ("rpcService"; "path");
    // returns "C:/Wakanda solutions/MySolution/MyProject1/MyProject1.waSettings"
```

getURLPath( )

Array **getURLPath**(String *url*)

Parameter	Type	Description
url	String	URL to parse
Returns	Array	Array of strings containing the parts of the URL

Description

The **getURLPath( )** method returns the *url* passed in the parameter as an array of strings.

The URL passed as a parameter is broken down according to the separators ("/" characters) that it contains. The values returned do not contain any "/" characters. For example, the URL "MyApp/Root/Manager/plan.html" returns the array ["MyApp","Root","Manager","plan.html"].

Note that the function detects "/" characters that are inserted into parameter strings (set between single or double quotes) and does not consider them as URL separators. For example, the following URL is divided into three parts: "MyApp/Root/Do_this("/T")"

If the URL contains a query string (placed after the "?" character), it is not parsed by the method. To parse this part of the URL, you must use the **getURLQuery( )** method.

getURLQuery( )

Object **getURLQuery**(String *url*)

Parameter	Type	Description
url	String	URL to parse

Returns	Object	Sub-queries of the URL's 'query string'

Description

The **getURLQuery( )** method returns in an object the contents of the *url*'s"query string", which was passed as a parameter. The query string is the part found after the "?" in the URL.

Each sub-query in the form "[&]a=1" generates a new member as follows: {a : 1}.

Example

You pass a complete URL to the method:

```
var theQuery = getURLQuery("http://127.0.0.1:8081/rest/Employees/?$top=40&$method=entityset&$timeout=300");
        // theQuery is {$top:40, $method:entityset, $timeout:300}
        // theQuery.$top is 40
```

getWalibFolder( )

Folder | String **getWalibFolder**([String *kind* [, Boolean | String *format*]])

Parameter	Type	Description
kind	String	Type in which the folder must be returned: Folder object (if omitted), path, relativePath, or url
format	Boolean, String	True or "posix" or "encoded" (default); False or "system" or "notEncoded"
Returns	Folder, String	Wakanda Server walib folder path or reference

Description

The **getWalibFolder( )** method returns Wakanda Server's "walib" folder, which contains the libraries and services available client-side, such as WAF and RpcService.

Note that this method is available in the Solution and Application classes: both return the same result unless you pass "relativePath" to the *kind* parameter.

You can get the folder in different formats depending on the string you pass to the *kind* parameter:

- If you omit the *kind* parameter, this method returns an object of type *Folder* (see the **Folder Class** methods and properties).
- If you pass "path" or "relativePath", this method returns a string containing the path to the folder. If you pass "relativePath", the path will be relative to the application or solution folder depending on the object to which it is applied.
  By default, the path is expressed in posix syntax. You can get this value in the system's syntax by passing *false* or "system" in the *format* parameter (you can set the standard format back by passing *true* or "posix").
- If you pass "url", the method returns a string containing the folder's URL.
  By default, the URL is encoded. You can get this value without encoding by passing *false* or "notEncoded" in the *format* parameter (you can set the standard format back by passing *true* or "encoded" in this parameter).

Note: On Mac OS, posix and system paths are equivalent.

Example

The following example illustrates the different values that can be returned by **getWalibFolder( )**:

```
var theEncodedURL=getWalibFolder("url"); // returns file:///C:/Wakanda%20versions/Wakanda%20Server/walib/
var theRawURL=getWalibFolder("url", false); // returns file:///C:/Wakanda versions/Wakanda Server/walib/
var thePath=getWalibFolder("path"); // returns C:/Wakanda versions/Wakanda Server/walib/
var theFolder=getWalibFolder(); // returns { name: "walib", extension: "", folders: Array(), 11 more}...
```

include( )

void **include**(File | String *file* [, String | Boolean *refresh*])

Parameter	Type	Description
file	File, String	JavaScript file to include in your code
refresh	String, Boolean	"refresh" or true = reevaluate files already included, "auto" or false = do not reevaluate files (default if omitted)

Description

The **include( )** method references a JavaScript *file* from a parent JavaScript file. Once the *file* is referenced, you can call and use the code and functions it contains just as if they were written in the parent file.

In *file*, pass either a *File* object or a string containing a valid path to the JavaScript file you want to reference.

By default, an included JavaScript file is evaluated only once: if several **include( )** methods are executed on the same *file* while the JavaScript context is alive, the file will not be reevaluated. This means that variables will keep their current value and will not be initialized.
If you want to force the included file to be reevaluated (and thus the variables to be initialized) each time the **include( )** method is called in the running JavaScript context, pass the "refresh" string or **true** in the *refresh* parameter. To use the default behavior, pass the "auto" string or **false**, or omit the *refresh* parameter.

Example

Imagine you wrote a utility function in a "myUtils.js" file (stored in a "ssjs" subfolder of your project folder):

```
 // code of myUtils.js file
var concatStr = function(inStr1, inStr2) {
    return inStr1 + " " + inStr2;
}
```

If you reference this file using the **include( )** method, you can call the function just as if it belongs to the one that is currently executed:

```
include(ds.getModelFolder().path + 'ssjs/myUtils.js');
// ...some code...
var aStr = concatStr("Hello","World!");
```

isoToDate( )

Date **isoToDate**(String *isoDate*)

Parameter	Type	Description
isoDate	String	String date in ISO format
Returns	Date	Date object

Description

The **isoToDate( )** methods converts the ISO date string passed in the *isoDate* parameter into a standard JavaScript format.
This method is designed to be used in conjunction with the **dateToIso( )** to manage dates in Wakanda's widgets.

Example

```
function getYear(isoDate) {
    return isoToDate(isoDate).getFullYear();
}
```

JSONToXml( )

String **JSONToXml**(String *jsonText*, String *jsonFormat*, String *rootElement*)

Parameter	Type	Description
jsonText	String	JSON formatted string
jsonFormat	String	JSON format to use (always "json-bag")
rootElement	String	Name of the root element to create
Returns	String	jsonText string converted to an XML string

Description

The **JSONToXml( )** method returns a JSON string converted into an XML string.

- Pass in the *jsonText* parameter a valid JSON string to convert.
- Pass in the *jsonFormat* parameter a string representing the JSON format to generate. In the current version of Wakanda, only the json-bag format is supported. So, you just need to pass the string "json-bag".
- Pass the name of the root element that you want to create in the resulting XML string to the *rootElement* parameter.

Note that since JSON and XML semantics are different and the resulting XML string may not reflect exactly the *jsonText* contents:

- Arrays of elements are not supported in XML. If the *jsonText* parameter contains such arrays, they are not converted. Only arrays of objects are supported.
- Since XML makes no distinction between single objects and arrays, the method adds the "____objectunic=true" attribute so that a reverse conversion (using the **XmlToJSON( )** method) can be done correctly.

Example

The following function converts any object into JSON then into XML before saving it in a text file:

```
function saveToXMLFile (object, root, file) {
    var xmlText, jsonText, root;
    jsonText = JSON.stringify(object);    // convert the object into JSON string
    xmlText = JSONToXml(jsonText, "json-bag", root);   // convert the JSON into XML
    saveText (xmlText, file);   // save the contents in an XML file
    }
```

- If you call the function with the following parameters:

```
saveToXMLFile (
{ a:[{e:12},{e:5},{e:7}], b:{x:"Monday", y:50}, c:"Tuesday" }, // the object to convert
"MyRoot", // the root element to add
"c:/temp/myXMLFile.xml"); // the path of the file to create
```

… you will create an XML file with the following contents:

```
<?xml version="1.0" encoding="utf-8" ?>
<MyRoot c="Tuesday">
  <a e="12" />
  <a e="5" />
  <a e="7" />
  <b ____objectunic="true" x="Monday" y="50" />
</MyRoot>
```

- If you call the function with the following parameters:

```
saveToXMLFile (
      {"__ENTITIES":
          {
          "__KEY":"1",
          "__STAMP":3,
          "uri":"http://127.0.0.1:8081/rest/Employee(1)",
          "ID":1,
          "Name":"Smith",
          "jobName":"Webmaster",
          "salary":20000
         }
      }, // the object to convert
      "MyEntity", // the root element to add
      "c:/temp/myXMLFile2.xml"); // path of the file to create
```

... you will create an XML file with the following contents:

```
<?xml version="1.0" encoding="utf-8" ?>
 <MyEntity>
  <__ENTITIES ____objectunic="true" __KEY="1"  __STAMP="3" uri="http://127.0.0.1:8081/res
</MyEntity>
```

loadImage( )

Image **loadImage**(File | String *file*)

Parameter	Type	Description
file	File, String	Image file object or path
Returns	Image	Image object

Description

The **loadImage( )** method loads the image stored in a file referenced by the *file* parameter and returns an *image* object. You can pass either a *File* object or a string containing a standard file path in the *file* parameter (use the "/" as folder separator).

Note (Beta): In the current version of Wakanda, you have to pass an absolute path in the file parameter.

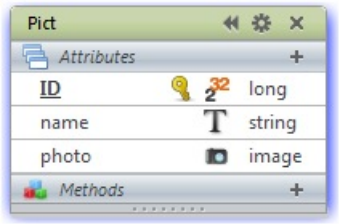
If the file does not contain a valid image or if the file reference is invalid, the method returns *null*.

For more information about Wakanda image object manipulation, refer to the **Images** documentation.

Example

This example loads the image in a JPG file stored on the server and stores it in a new entity in the Pict class (in the photo attribute).

Here is the (simplified) datastore class:




```
var mypict = loadImage ("C:/Wakanda/Solutions/mysolution/Tulips.jpg"); // load the image from file
var p = new Pict(); // create a new entity in the Pict datastore class
p.name = "Flower"; // name the image
p.photo = mypict; // put the image in the photo attribute
p.save(); // save the entity
```

loadText( )

String **loadText**(File | String *file* [, Number *charset*])

Parameter	Type	Description
file	File, String	Text file object or path
charset	Number	Character set of the text
Returns	String	String loaded from the file parameter

Description

The **loadText( )** method loads the text stored in a file referenced by the *file* parameter and returns a string containing the text. You can pass either a *File* object or a string containing a standard file path in the *file* parameter (use the "/" as folder separator).

Note: In the current version of Wakanda, you have to pass an absolute path in the file parameter.

You can pass a code to indicate the charset of the loaded text in the *charset* parameter. By default, if the parameter is not passed, Wakanda uses the UTF-8 charset (value = 7). To get a list of available charset codes, see the **TextStream( )** method from the **Files and Folders** documentation.

Example

We want to load a text file contents in a variable. The text file is located in a subfolder named "Import" within the model folder:

```
var info = loadText(ds.getModelFolder().path + "Import/info.txt");
```

loginByKey( )

Boolean **loginByKey**(String *name* , String *key* [, Number *timeOut*])

Parameter	Type	Description
name	String	User name
key	String	HA1 key of the user
timeOut	Number	User session timeout (in seconds)
Returns	Boolean	True if the user has been successfully logged, otherwise False

Description

Note: In Wakanda v1, this method belongs to the Datastore class.

The **loginByKey( )** method authenticates a user by their *name* and HA1 *key* and, in case of success, opens a new user session on the server. To be validated, both *user* and *key* must be registered in the directory of the application (for more information, please refer to the **Users and Groups**) section.
If the authentication is completed successfully, the method returns **true** and opens a user session on the server.

In *name*, pass a string containing the name of the user you want to log in.

In *key*, pass the HA1 hash key of the user you want to log in. The HA1 key results from a combination of several information, including the name and the password of the user, using a hash function. This key has to be generated on the client-side using a specific method *(Implementation Note: This method is currently being developed)*.

In *timeout*, pass a value in seconds to set the user session timeout, i.e., the time the server will keep the user session open if no user query has been received. By default, if you omit this parameter, the user session timeout is 3600 (i.e., one hour).

loginByPassword( )

Boolean **loginByPassword**(String *name* , String *password* [, Number *timeOut*])

Parameter	Type	Description
name	String	User name
password	String	User password
timeOut	Number	User session timeout (in seconds)
Returns	Boolean	True if the user has been successfully logged, otherwise False

Description

Note: In Wakanda v1, this method belongs to the Datastore class.

The **loginByPassword( )** method authenticates a user by their *name* and *password* and, in case of success, opens a new user session on the server. This method is designed to be used when the solution authentication mode is "custom", that is, when you manage the identification through your code (for more information about authentication modes, please refer to paragraph **Setting the Authentication Mode**).

To be validated, both *user* and *password* must be registered in the directory of the application (for more information, please refer to the **Users and Groups** section).

If the authentication is completed successfully, the method returns **true** and opens a user session on the server.

In *name*, pass a string containing the name of the user you want to log in.

In *password*, pass the password of the user you want to log in. Note that the password comparison is case sensitive.

In *timeout*, pass a value in seconds to set the user session timeout, i.e., the time the server will keep the user session open if no user query has been received. By default, if you omit this parameter, the user session timeout is 3600 (i.e., one hour).

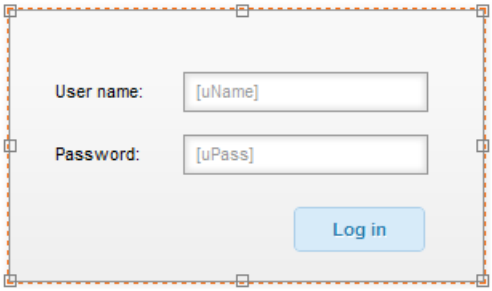
Example

We want the users to connect using a standard login area in our Web pages.

1. The current authentication mode is "custom", that is, the following line has been defined in the {*solutionName*}.waSettings file:

   ```
   <directory authenticationType=""/>
   ```

2. On the client side, we designed the following login area in one of the pages of our Web application:



Both *uName* and *uPass* are local datasources (based on variables).
The script for the **Log in** button is the following:

```
button2.click = function button2_click (event)
{
   ds.Person.login(uName, uPass); // just call the 'login' datastore class method with user ids
};
```

3. On the server, we wrote the following 'login' datastore class method (applied to the Person class):

```
model.Person.methods.login = function(userName, password) // the function gets name and password
{
   var result = ds.loginByPassword(userName, password, 60*60); // session is created in case of success
   return result; // result is sent to the client
}
```

logout( )

Boolean **logout**( )

Returns	Boolean	true if the user has been successfully loggued out

Description

Note: In Wakanda v1, this method belongs to the Datastore class.

The **logout( )** method logs out the user running the current session on the Wakanda server. After the method is executed, there is no user loggued in the thread running the script.

If the user logout is executed successfully, the method returns **true**.

open4DBase( )

Datastore **open4DBase**(File *structure4D*, File *data4D*)

Parameter	Type	Description
structure4D	File	Reference to a 4D database structure file
data4D	File	Reference to a 4D database data file
Returns	Datastore	Temporary datastore object

Description

You use this method to create a temporary internal Wakanda datastore based on the 4D database designated by the *structure4D* and *data4D* parameters.

Once the datastore is created, you can work with it just like you do with any other datastore and perform queries, sorts, and so on. You can also modify the data or the model using the API commands. All these modifications are stored directly in the 4D database. The 4D database can then be opened again in 4D. This command supports 4D v12 databases and higher.

Warning: As the 4D database is open in read/write mode by open4DBase( ), it must not be already opened by 4D or 4D Server.

- Pass the reference to a valid 4D structure file in the *structure4D* parameter (file ending with .4db).
- Pass the reference to a valid 4D data file in the *data4D* parameter (file ending with .4dd).

Note: You can get file references using the File( ) method. For more information, refer to the description of the Files and Folders.

This method returns a reference of type *Datastore* to the datastore that you opened. You can assign this reference to a variable that you can then use as the target object for queries, just like you do with the **ds** object. If you use a variable named *ds*, it will override the current datastore.

To close the datastore and thus the database opened with this method, do not forget to eventually call the **close( )** method.

openDataStore( )

Datastore **openDataStore**(File *model*, File *data*)

Parameter	Type	Description
model	File	Reference to an existing model file
data	File	Reference to an existing data file
Returns	Datastore	Reference to the open datastore object

Description

The **openDataStore( )** method opens a datastore in the current solution and returns a reference to it.

- Pass the reference of an existing datastore model file (file ending with .waModel) on disk to the *model* parameter.
- Pass in *data* the reference of a data file (file ending with .waData) that corresponds to the datastore model. Unlike the **createDataStore( )** method, the **openDataStore( )** method requires that the data file already physically exists on the server.

Note: For more information about the File( ) method, refer to the description of Files and Folders.

This method returns a *Datastore* object type reference to the datastore that you opened. You can assign this reference to a variable that you can then use as the target object for queries, just like you use the **ds** object (see example below). If you use a variable named *ds*, it will override the current datastore.

To close the datastore opened with this method, do not forget to call the **close( )** method eventually.

Example

The following example opens an additional datastore and performs a query on the data in this datastore:

```
var modelFile= File("c:/myfolder/myDS.waModel");     // get a reference to the datastore model file
var dataFile= File("c:/myfolder/myDS.waData");     // get a reference to the data file
if (modelFile.exists && dataFile.exists)     // if both files actually exist on the disk
{
    var myDS = openDataStore(modelFile, dataFile);     // open and reference the datastore
    var mySet = myDS.People.query( "name = :1", "Jones");      // perform a query on the datastore
```

```
}
```

ProgressIndicator( )

ProgressIndicator **ProgressIndicator**(Number *numElements* [, String *sessionName*][, Boolean | String *stoppable*][, String *unused*[, String *name*]])

Parameter	Type	Description
numElements	Number	Number of elements to count
sessionName	String	Name of execution session for progress indicator
stoppable	Boolean, String	True or "Can Interrupt" = Progress indicator can be stopped false or "Cannot Interrupt" = Progress indicator cannot be stopped
unused	String	Not used, always pass an empty string ("")
name	String	Unique name of object on the server
Returns	ProgressIndicator	New progress indicator created on the server

Description

The **ProgressIndicator( )** method can be seen as the constructor of the **ProgressIndicator** class. It creates a new object of type *ProgressIndicator* on the server and specifies its properties through several parameters:

- *numElements*: In this parameter, you pass the number of elements whose processing progress must be shown. For example, if the operation associated with the progressIndicator performs processing on 10,000 entities, you pass 10000 to this parameter. The processing progress through these 10,000 entities is transmitted to the object using the **setValue( )** and **incValue( )** methods. The session is terminated when this maximum is reached.
  If you pass -1 in *numElements*, you create an "infinite" type progressIndicator.
- *sessionName*: In this parameter, you pass the name of the progressIndicator object's execution session. The execution session is the period during which the progressIndicator is active. By default, the string "Progress bar on [*Progress Reference Name*]" is used by the GUI Designer. You can modify this string as desired.
  You can use the following placeholders in *sessionName* for the progressIndicator to display additional information:
  - {curValue} is replaced by the current element in *numElements* that is currently being processed.
  - {maxValue} is replaced by the *numElements* value.

  This lets you display, for instance, "Processing the {curValue} entity out of {maxValue}".
  Note: This parameter can also be set dynamically by using the **setMessage( )** *method.*
- *stoppable*: This parameter indicates whether the progressIndicator can be interrupted by the user. Pass *true* or the string "Can Interrupt" if you want the user to be able to stop it. Otherwise, pass *false* or the string "Cannot Interrupt".
- *unused*: This parameter is reserved, so just pass an empty string "" to it.
- *name*: This parameter must contain the unique reference of the progressIndicator object on the server. You use this reference to associate client widgets with progressIndicators that are executed on the server. On the client, this parameter must correspond to the **Progress Reference** field defined for the Display Error widget in the GUI Designer.

Note that this method creates an object and executes a progress session on the server, but does not support its display on the client. In your client-side interface, you must add a Display Error widget attach it to the server session (see example below).

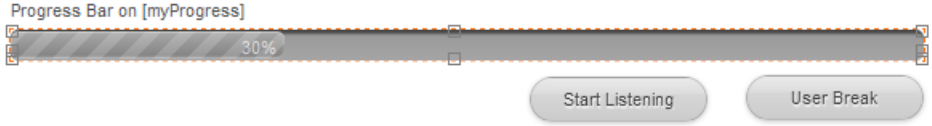
Example

In this example, we create a progressIndicator on the server (based on a basic processing task) and associate a Progress Bar widget with it on the client.

- Here is the server-side code:

```
// create a progress indicator object and open the session
var prog = ProgressIndicator(1000000, "Processing element {curValue} out of {maxValue}", true, "", "myProgı
var s = "" ;
for (var i = 1; i  < 1000000; i++) // Processing loop
{
    if (prog.setValue(i))  // Increment progress indicator and test for a possible interruption
    {
        s += i; // Processing
    }
    else // If there is a user interrupt request
        break;
}
prog.endSession();  // Close the progressIndicator session
```

- On the client, the following widget is added in the GUI Designer:



  It has the following properties:
  - *ID* = "progressBar0"
  - *Progress reference* = "myProgress"

  You can manage the connection between the widget and the server session through Button widgets:
  - **Start Listening** begins sending requests to the server in order to display the progress of the session associated with the widget. The following code is associated with this button:

```
button0.click = function button0_click (event)
{
    $$("progressBar0").startListening(); // connect to the session on the server
};
```

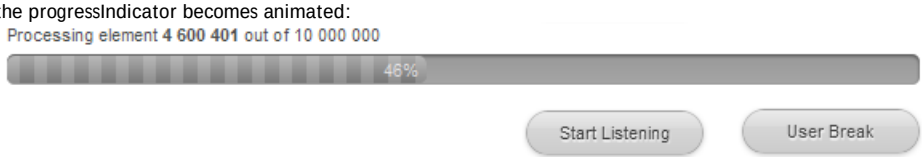
Note: Once connected to the session on the server, the widget regularly sends repeated requests to the server to get the status of the progressIndicator. This is why the connection to the session must be explicitly enabled using the startListening() method so as to avoid

unnecessary requests. In our example, this method is placed in the code of the button for simplicity's sake; usually, the starting code can be called directly in the requests initiating processing on the server.

- **User Break** sends a request to the server to interrupt the session. If the progressIndicator is "stoppable", the **setValue( )** or **incValue( )** method returns *false* at the next test and you can stop its execution (for example, using the **break** method as seen above). The following code is associated with this button:

```
button0.click = function button0_click (event)
 {
     $$("progressBar0").userBreak(); // send request to interrupt
 };
```

- You can then execute the code on the server and connect to it through your Interface page. As soon as you click on the **Start Listening** button, the progressIndicator becomes animated:



Note: Keep in mind that the client-side progressIndicator is associated with the progressIndicator on the server but both objects are independent. If you exit the browser, the progressIndicator session on the server continues. If you reconnect and start listening again, the client-side progressIndicator immediately displays the current percentage of progress.

removeHttpRequestHandler( )

void **removeHttpRequestHandler**(String *pattern*, String *filePath*, String *functionName*)

Parameter	Type	Description
pattern	String	Request pattern to remove
filePath	String	Path to the file in which the handler function is defined
functionName	String	Name of the function to handle the request matching the pattern

Description

The **removeHttpRequestHandler( )** method uninstalls an existing HTTP request handler function running on the server. The request handler should have been installed using the **addHttpRequestHandler( )** method.

For a complete description of the server-side HTTP request handlers feature, refer to the **HTTP Server Request Handlers** documentation.

require( )

Module **require**(String *id*)

Parameter	Type	Description
id	String	Module identifier
Returns	Module	CommonJS compliant module

Description

The **require( )** method returns the exported API of a CommonJS compliant *Module* whose identifier you pass in *id*.

For more information about the *id* parameter (module identifier) and the CommonJS architecture, please refer to the CommonJS specification.

Wakanda CommonJS modules must be stored in a **'modules**' folder located at the root of the application folder.

Wakanda proposes various built-in CommonJS utility modules, for example for handling network connections. These modules as well as their APIs are described in the **SSJS Modules** section.

Example

This example from the CommonJS specification shows a 'call chain' of modules.

- The "math.js" file is located in the **modules** folder:

```
exports.add = function() {
    var sum = 0, i = 0, args = arguments, l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum;
};
```

- The "increment.js" file is also located in the **modules** folder:

```
var add = require('math').add;
 exports.increment = function(val) {
     return add(val, 1);
 };
```

- This code can be executed in any server-side .js file of the project:

```
var inc = require('increment').increment;
var a = 1;
inc(a); // 2
```

saveText( )

void **saveText**(String *textToSave* , File | String *file* [, Number *charset*])

Parameter	Type	Description
textToSave	String	Text to be saved
file	File, String	Text file object or path
charset	Number	Character set of the text

Description

The **saveText( )** method saves the text you passed to the *textToSave* parameter in the *file* parameter. You can pass either a *File* object or a string containing a standard file path in the *file* parameter (use the "/" character as the folder separator).

Note (version 0.5): In the current version of Wakanda you have to pass an absolute path to the file parameter.

You can pass a code to indicate the charset of the loaded text in the *charset* parameter. By default, if the parameter is not passed, Wakanda uses the UTF-8 charset (value = 7). To get a list of available charset codes, see the TextStream class from the **Files and Folders** documentation.

setInterval( )

Number **setInterval**(Function *handler* [, Number *timeout* [, Mixed *args*,..., Mixed *args*N]])

Parameter	Type	Description
handler	Function	A handler to be executed every timeout milliseconds
timeout	Number	Timeout in milliseconds
args	Mixed	Argument(s) to pass to the handler
Returns	Number	timerID handle

Description

The **setInterval( )** method schedules JavaScript code to be run every *timeout* milliseconds.

Pass in *handler* the function to be run periodically and in *timeout* the time lapse in milliseconds to wait between two run. Note that the *timeout* is optional. If you omit this parameter, a 10 ms default timeout will be applied. This default value is also applied if you pass a *timeout* less than 10 ms.

The optional *args* parameter(s) are passed directly to the given *handler*.

The **setInterval( )** method returns a *timerID* which can be used in a subsequent call to **clearInterval( )**. Scheduled code can be canceled using the **clearInterval( )** method.

Note that callbacks (scheduled code) cannot be executed at the same time as other JavaScript code. The **wait( )** method must be called to handle asynchronous execution.

Note: The Wakanda **setInterval( )** method is compliant with the Timers W3C specification.

Example

In this example, a code will be called every second during 5 seconds and then will wait 5 seconds before stopping:

```
var n = 5;
var i = 1;
var id;

function delayedStop()
{
    // Close the worker
    close();
}

function periodicFunction()
{
        // Callbacks
    if (++i == n + 1)
    {
        clearInterval(id);
           // Waiting 5s before quitting
        setTimeout(delayedStop, 5000);
    }
}

    //Start periodic callback function
var id = setInterval(periodicFunction, 1000);
wait();
```

setTimeout( )

Number **setTimeout**(Function *handler* [, Number *timeout* [, Mixed *args*,..., Mixed *args*N]])

Parameter	Type	Description
handler	Function	Any handler whose execution must be delayed
timeout	Number	Timeout in milliseconds
args	Mixed	Argument(s) to pass to the handler
Returns	Number	Timeout handle

Description

The **setTimeout( )** method allows you to schedule JavaScript code to be executed after a given delay.

Pass in *handler* the function to be delayed and in *timeout* the delay to apply in milliseconds. Note that the *timeout* is optional. If you omit this parameter, a 4 ms default timeout will be applied. This default value is also applied if you pass a *timeout* less than 4 ms.

The optional *args* parameter(s) are passed directly to the given *handler*.

The **setTimeout( )** method returns a *timerID* which can be used in a subsequent call to **clearTimeout( )**. Pending scheduled code can be canceled using the **clearTimeout( )** method.

Note that callbacks (scheduled code) cannot be executed at the same time as other JavaScript code. The **wait( )** method must be called to handle asynchronous execution.

Note: The Wakanda **setTimeout( )** method is compliant with the Timers W3C specification.

Example

He is a typical structure that can be written using this method:

```
function delayedStart ()
{
    ... // code to be started after 5 seconds
    close(); // to free up memory
}

    // Code to start in 5 seconds
setTimeout(delayedStart, 5000);
wait();
```

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. It allows you to create new **Shared Web Workers (parent)** objects on the server.

Shared workers are Web workers that can be addressed from any thread, while dedicated workers are Web workers that can only be addressed from the parent thread that created them. Dedicated workers end when the parent thread ends while shared workers continue to exist even if their spawning thread ends. For more information, refer to the **Dedicated Web Workers (parent)** class description.

Shared workers are uniquely identified by their script file names and a given name. The constructor will spawn a new shared worker thread if it does not exist yet.

In the *scriptPath* parameter, pass a path to a project-specific JavaScript file. If you pass the file with a relative path, Wakanda assumes the project folder as the default. The referenced file must have valid statements that result in a worker.

Note: If the worker's JavaScript file has any code outside of all function declarations, Wakanda considers it initialization code for the worker and executes it when the worker is created.

In *workerName*, pass the name of the shared worker you want to create (if you omit the *workerName* parameter, the shared worker will be created with an empty string as its name). This shared worker name will be used to reference the shared worker for all the threads. When other threads want to interact with an already existing shared worker, they do so by executing the same code as if they are creating it, but instead receive a reference to the existing shared worker.

Example

This shared worker will create an entity every second for 5 seconds, and sends infos to the log. Here is the launcher function:

```
function doTestSharedWorker()
{
      var theWorker = new SharedWorker("SendRequestsWorker.js", "SendRequests");
      var thePort = theWorker.port; // MessagePort
      thePort.onmessage = function(evt)
      {
            var message = evt.data;
            switch(message.type)
            {
                  case 'error':
                        debugger;
                        break;
            }
      }
}
doTestSharedWorker();
```

Here is the code of the SendRequestsWorker.js file:

```
function doSendRequests()
{
      count++;
      console.log('Count: ' + count);

      var theDate = new Date();
      if((theDate - startDate) < theDuration) {
            console.log('creating');
            var z = new ds.Util({
                        testValue    : count,
                        dateValue     : theDate
                  });
            z.save();
            console.log('' + ds.Util.length);
```

```
      } else {
            console.log('closing');
            close();
      }
}

onconnect = function(msg)
{
    var thePort = msg.ports[0];
    console.log(In onconnect');
    thePort.postMessage("OK");
}
console.log('Start of test...');

var count = 0;
var startDate = new Date();
var theDuration = 5000;

setInterval(doSendRequests, 1000) //Run every second
```

Example

Here is a basic example of creating a shared worker: the purpose of this datastore class method is to respond to a browser-side request for information on the status of the "TaskMgr" shared worker.

```
getTaskManagerStatus:function()
{
var tmRef = 0;
var tmInfo = {taskCount:0, errorCode:0};
var taskMgr = new SharedWorker('WorkersFolder/TaskMgr.js', 'TaskMgr');
var thePort = taskMgr.MessagePort;
thePort.onmessage = function(event)
{
    var message = event.data;
    switch (message.type)
    {
        case 'connected':
            tmRef = message.ref;
            thePort.postMessage({type: 'report', ref: tmRef});
            break;

        case 'update':
            taskMgrInfo.taskCount = message.count;
            return taskMgrInfo;
            thePort.postMessage({type: 'disconnect', ref: tmRef});
            close();
            break;

        case 'error':
            taskMgrInfo.errorCode = message.errorCode;
            return taskMgrInfo;
            close();
            break;
        }
    }
    wait();
}
```

The corresponding "TaskMgr" worker might be something like this:

```
function doSomeWork()
{
    try {
        // do something
        tmCount += 1;
    }
    catch(e){
        tmError = 1;
    }
}

onconnect = function(msg) // called when a new SharedWorker is created
{
    var thePort = msg.ports[0];
    tmKey += 1;
    tmConnections[tmKey] = thePort;
    thePort.onmessage = function(event)
    {
        var message = event.data;
        var fromPort = tmConnections[message.ref];
        switch (message.type)
        {
            case 'report':
                if (tmError!= 0)
                {
                    fromPort.postMessage({type: 'error', errorCode: tmError });
                    close();
```

```
                }
                else
                {
                    fromPort.postMessage({type: 'update', count: tmCount});
                }
                break;

            case 'disconnect':
                tmConnections[message.ref] = null;
                break;
        }
    }
    thePort.postMessage({type: 'connected', ref: tmKey});
}

var tmCount = 0;
var tmKey = 0;
var tmError = 0;
var tmConnections = [];
setInterval(doSomeWork(), 1000) //Run every second
```

SystemWorker( )

void **SystemWorker**(String *commandLine*)

Parameter	Type	Description
commandLine	String	Command line to execute

Description

The **SystemWorker( )** method is the constructor of the *SystemWorker* type class objects.

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

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

In the *commandLine* parameter, pass the application's absolute file path to execute, as well as any required arguments (if necessary). Under Mac OS, if you pass only the application name, Wakanda will use the PATH environment variable to locate the executable.

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

Example

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

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

TextStream( )

TextStream **TextStream**(String | File *file* , String *readMode* [, Number *charset*])

Parameter	Type	Description
file	String, File	Binary text file to reference
readMode	String	Streaming action: "Write" to write data, "Read" to read data, "Overwrite" to replace the file with new data
charset	Number	Character set of the text (7 i.e. UTF-8 by default)
Returns	TextStream	New TextStream object

Description

The **TextStream( )** method creates a new *TextStream* object. *TextStream* objects are handled using the various properties and methods of the **TextStream Class**.

In the *file* parameter, pass the path of the text file or a reference to it. The value can be either:

- an absolute path (using the "/" separator) or a URL, including the file name or
- a valid *File* object

Once the file is referenced, you can start writing or reading the stream data depending on the value you passed in the *readMode* parameter:

- If you passed "Write", the file is opened in write mode.
- If you passed "Read", the file is opened in read mode.
- If you passed "Overwrite", the file is replaced by the data you will write.

The *charSet* parameter is optional. It can be used to indicate a charset that is different from the default one (UTF-8). This parameter takes an integer as a value. Setting a charset overrides the default charset unless a BOM is detected in the text in which case the BOM's charset is used. Here is a list of the most common accepted values:

- -2 - ANSI
- 0 - Unknown
- 1 - UTF-16 Big Endian
- 2 - UTF-16 Little Endian
- 3 - UTF-32 Big Endian
- 4 - UTF-32 Little Endian
- 5 - UTF-32 Raw Big Endian
- 6 - UTF-32 Raw Little Endian
- 7 - UTF-8

- 8 - UTF-7
- 9 - ASCII
- 10 - EBCDIC
- 11 - IBM code page 437
- 100 - Mac OS Roman
- 101 - Windows Roman
- 102 - Mac OS Central Europe
- 103 - Windows Central Europe
- 104 - Mac OS Cyrillic
- 105 - Windows Cyrillic
- 106 - Mac OS Greek
- 107 - Windows Greek
- 108 - Mac OS Turkish
- 109 - Windows Turkish
- 110 - Mac OS Arabic
- 111 - Windows Arabic
- 112 - Mac OS Hebrew
- 113 - Windows Hebrew
- 114 - Mac OS Baltic
- 115 - Windows Baltic
- 116 - Mac OS Simplified Chinese
- 117 - Windows Simplified Chinese
- 118 - Mac OS Traditional Chinese
- 119 - Windows Traditional Chinese
- 120 - Mac OS Japanese
- 1000 - Shift-JIS (Japan, Mac/Win)
- 1001 - JIS (Japan, ISO-2022-JP, for emails)
- 1002 - BIG5, Chinese (Traditional)
- 1003 - EUC-KR, Korean
- 1004 - KOI8-R, Cyrillic
- 1005 - ISO 8859-1, Western Europe
- 1006 - ISO 8859-2, Central/Eastern Europe (CP1250)
- 1007 - ISO 8859-3, Southern Europe
- 1008 - ISO 8859-4, Baltic/Northern Europe
- 1009 - ISO 8859-5, Cyrillic
- 1010 - ISO 8859-6, Arab
- 1011 - ISO 8859-7, Greek
- 1012 - ISO 8859-8, Hebrew
- 1013 - ISO 8859-9, Turkish
- 1014 - ISO 8859-10, Nordic and Baltic languages (not available on Windows)
- 1015 - ISO 8859-13, Baltic Rim countries (not available on Windows)
- 1016 - GB2312, Chinese (Simplified)
- 1017 - GB2312-80, Chinese (Simplified)
- 1018 - ISO 8859-15, ISO-Latin-9
- 1019 - Windows-31J (code page 932)

Example

We want to implement a Log function that we could call to create new log files and append messages at any moment. Using text streams is very useful in this case:

```
function Log(file)  // Constructor function definition
{
    var log =
    {
        appendToLog: function (myMessage)   // append function
        {
            var file = this.logFile;
            if (file != null)
            {
                if (!file.exists)  // if the file does not exist
                    file.create();  // create it
                var stream = TextStream(file, "write");  // open the stream in write mode
                stream.write(myMessage+"\n"); // append the message to the end of stream
                stream.close(); // do not forget to close the stream
            }
        },

        init: function(file)  // to initialize the log
        {
            this.logFile = file;
            if (file.exists)
                file.remove();
            file.create();
        },

        set: function(file)  // to create the log file
        {
            if (typeof file == "string") // only text files can be created
                file = File(file);
            this.logFile = file;
        },

        logFile: null
    }

    log.set(file);

    return log;
```

```
}
```

We can then create any log file we want and add messages in a very simple way, for example:

```
var log = new Log("c:/wakanda/mylog.txt"); // Creates a log file
var log2 = new Log("c:/wakanda/mylog2.txt"); // Creates another log file
log.appendToLog("*** First log file header***");
log2.appendToLog("*** Second log file header***");
log.appendToLog("First log entry in log1");
log2.appendToLog("First log entry in log2");
```

Example

We create a *TextStream* object using the constructor syntax and fill it byte by byte with the contents of a ANSI-encoded file:

```
var mystream = new TextStream("c:/temp/data.txt","Read",-2);
var data = "";
do
    {
    data = data + mystream.read(1);
    }
while(mystream.end()==false)
mystream.close();
```

wait( )

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

Parameter	Type	Description
timeout	Number	Timeout in milliseconds
Returns	Boolean	True if the worker is terminated. Otherwise, it is false.

Description

The **wait( )** method allows a thread to handle events and to continue to exist after the complete code executes.

In the context of a Web worker, the **wait( )** method allows a parent Web worker thread to handle child worker events. Since the parent-child worker communication is asynchronous (based on callbacks), this method is necessary in the parent script to allow the thread to keep from terminating after the code execution and to listen for callbacks. During the waiting time, asynchronous callback events from Web workers are handled. When this method has been called, the thread stays alive until you call **close( )**.

Note: The wait( ) method is also available for child workers although it is usually not necessary in this context. Child worker scripts always implicitly call the wait mechanism.

The **wait( )** method can also be used in the context of the main thread to allow asynchronous communication, for example when using **Net - Module** or **System Workers**. In this context, to stop the **wait( )** loop, you need to use **exitWait( )**.

Note that while executing, the **wait( )** method blocks the thread but still handles callbacks.

If you specify a value (in milliseconds) in the optional *timeout* parameter, **wait( )** will run only during the time specified and give the control back after this time, returning *false* if the worker is not terminated.

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*. It allows you to create new **Dedicated Web Workers (parent)** objects on the server. The proxy object allows the parent to exchange data with a dedicated worker.

Dedicated workers are Web workers that can only be addressed from the parent thread that created them while Shared workers are Web workers that can be addressed from any thread. Dedicated workers end when the parent thread ends while shared workers continue to exist even if their spawning thread ends. For more information, refer to the **Dedicated Web Workers (parent)** class description.

In the *scriptPath* parameter, pass a path to a project-specific JavaScript file. If you pass the file with a relative path, Wakanda assumes the project folder as the default. The referenced file must have valid statements that result in a worker.

Note: If the worker's JavaScript file has any code outside of all function declarations, Wakanda considers it initialization code for the worker and executes it when the worker is created.

Example

The following is a simple example of parent and child exchanging messages. Below is the **parent.js** script:

```
// A dedicated web worker is created by calling the Worker constructor
// with the name of the JavaScript file to execute (located in the default folder).
// This will return a proxy object (worker) so it will be possible to communicate with the child.
var worker = new Worker('child.js');

// Define the message callback that will be triggered each time the child sends a message
var state = 0;
worker.onmessage = function (event)
{
    if (state == 0) {
            // Child has received our initial message and this is its reply.
        console.log(event.data);    // "Child started".
```

```
            // Send a message to request termination
        worker.postMessage('Please quit.');
            // Go back to idle, waiting for reply from child.
        state = 1;

    } else {    // state == 1
            // Child has terminated
        console.log(event.data);    // Child finished
            // We can terminate by calling close(), which will exit the wait()
        close();
    }
}

// Send a message to the child to trigger message exchange
worker.postMessage("Go ahead.");

// Asynchronous execution
wait();

// After close() is called in callback, we are done.
console.log('Parent has terminated.');
```

Here is the **child.js** script:

```
// Child execution is asynchronous.
// onmessage is a global attribute containing the callback to trigger each time a message is received.
var state = 0;

onmessage = function (event) {
    if (state == 0) {
            // Waiting for a message from parent, just received it.
        console.log(event.data);    // "Go ahead".
            // Reply to parent. Note that postMessage() is a global method. It will send a message to the parent
            // worker proxy object onmessage attribute.
        postMessage("Child started");

            // Go back to idle, waiting for next message.
        state = 1;

    } else {    // state == 1
            // Waiting for a message from parent to terminate, just received it.
        console.log(event.data);    // "Please quit".
            // Sends a message back to parent, we're done.
        postMessage("Child finished");
            // Terminate.
        close();
    }
}
```

Note that there is no call to **wait( )**. By default, the child will wait until the end of the script, and service asynchronous callbacks.

In this example, the JavaScript code does nothing except for defining a callback. Everything is done by the callback.

XMLHttpRequest( )

void **XMLHttpRequest**([Object *proxy*])

Parameter	Type	Description
proxy	Object	Object containing a host and a port attributes

Description

The **XMLHttpRequest( )** method is the constructor of the class objects of the *XMLHttpRequest* type. It should be used with the **new** operator to create *XMLHttpRequest* instances on the server. Once created, an instance can be managed using methods and properties regarding the request itself (see **XMLHttpRequest Instances (Requests)**) as well as the response (see **XMLHttpRequest Instances (Responses)**).

If the Wakanda Server needs to perform the request through a proxy server, pass in the *proxy* parameter an object containing two attributes:

- host (string): address of the proxy server
- port (number): TCP port number of the proxy server

For example, this object is a valid *proxy* parameter:

```
{port: "http://proxy.myserver.com", host: 80}
```

Example

In the following example, we send GET requests to a Wakanda server or to an HTTP server and format the responses, whatever their type (HTML or JSON). We can then see the results in the Wakanda code editor.

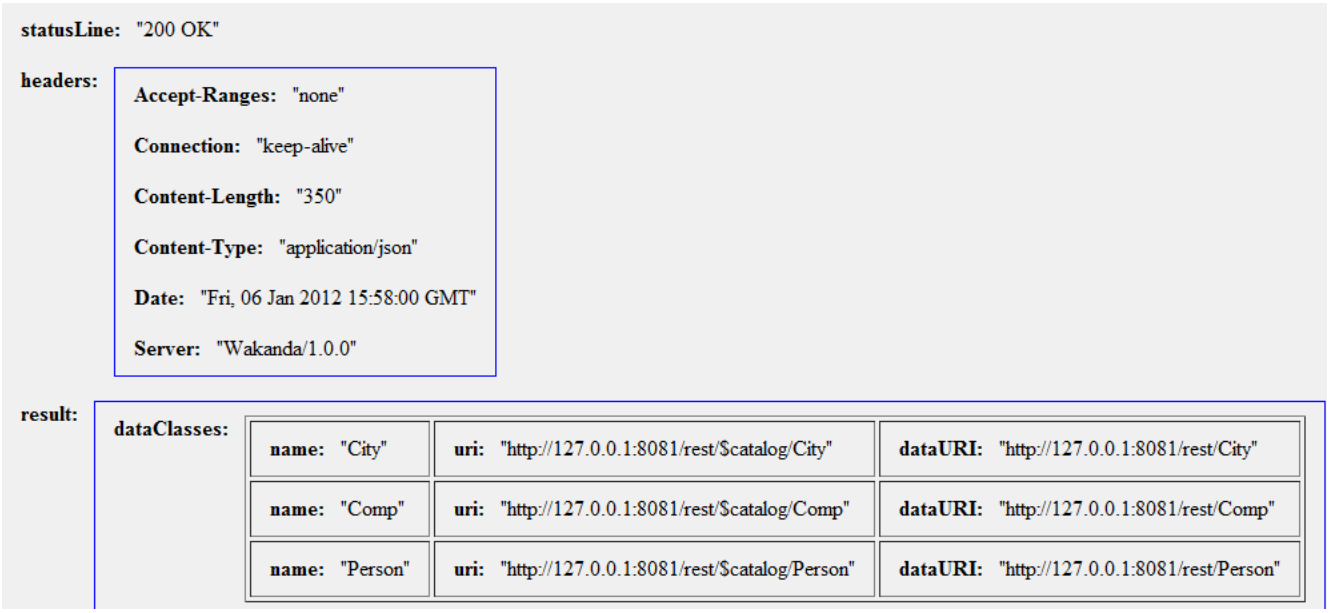
```
var xhr, headers, result, resultObj, URLText, URLJson;
 var proxy = { // define a proxy only if necessary
     host: 'proxy.myserver.com', // use any valid proxy address
     port: 80
 }

 URLJson = "http://127.0.0.1:8081/rest/$catalog"; // REST query to a Wakanda server
 URLText = "http://communityjs.org/"; // connect to an HTTP server
```

```
var headersObj = {};

xhr = new XMLHttpRequest(proxy); // instanciate the xhr object
   // the proxy parameter may not be necessary

xhr.onreadystatechange = function() { // event handler
    var state = this.readyState;
    if (state !== 4) { // while the status event is not Done we continue
        return;
    }
    var headers = this.getAllResponseHeaders(); //get the headers of the response
    var result = this.responseText;  //get the contents of the response
    var headersArray = headers.split('\n'); // split and format the headers string in an array
    headersArray.forEach(function(header, index, headersArray) {
        var name, indexSeparator, value;

       if (name.indexOf('HTTP/1.1') === 0) { // this is not a header but a status
            return; // filter it
        }

       indexSeparator = header.indexOf(':');
       name = header.substr(0,indexSeparator);
       if (name === "") {
           return;
       }
       value = header.substr(indexSeparator + 1).trim(); // clean up the header attribute
       headersObj[name] = value; // fills an object with the headers
    });
    if (headersObj['Content-Type'] && headersObj['Content-Type'].indexOf('json') !== -1) {
            // JSON response, parse it as objects
        resultObj = JSON.parse(result);
    } else { // not JSON, return text
        resultTxt = result;
    }
};

xhr.open('GET', URLText); // to connect to a Web site
  // or xhr.open('GET', URLJson) to send a REST query to a Wakanda server

xhr.send(); // send the request
statusLine = xhr.status + ' ' + xhr.statusText; // get the status

// we build the following object to display the responses in the code editor
({
    statusLine: statusLine,
    headers: headersObj,
    result: resultObj || resultTxt
});
```

The results can be displayed in the Results area of the Wakanda Studio code editor.

Here is the result of a simple query to an Web server:

Here is the result of a REST query to an Wakanda server:



XmlToJSON( )

String **XmlToJSON**(String *xmlText* , String *jsonFormat* [, String *rootElement*])

Parameter	Type	Description
xmlText	String	XML formatted string
jsonFormat	String	JSON format to use (always "json-bag")
rootElement	String	Name of the root element
Returns	String	xmlText string converted to JSON string

Description

The **XmlToJSON( )** method returns an XML string converted into a JSON string.

- Pass in the *xmlText* parameter a valid XML string to convert.
- Pass in the *jsonFormat* parameter a string representing the JSON format to use. In the current version of Wakanda, only the json-bag format is supported. At this time, just pass "json-bag" to this parameter.

- Pass the name of the root element of the XML string in the *rootElement* parameter.

Example

In this example, the model file of a Wakanda project (stored in XML format) is converted into JSON:

```
var xmltext = loadText("C:/Wakanda/test/test.waModel");
// open and get the contents of the model file
var jsonText = XmlToJSON(xmltext, "json-bag", "EntityModelCatalog");
// convert the XML into JSON
var mymodel = JSON.parse(jsonText);
// return a JavaScript object from the JSON object
```

HTTP Server

The HttpServer class provides properties and methods for managing the Wakanda built-in HTTP server.

You can use these properties and methods through the global application **httpServer** property.

cache

Description

The **cache** property is an object of type *HttpServerCache* that contains two properties:

- **enabled** - Boolean: The status of the HTTP server cache.
- **memorySize** - Number: The size of the HTTP server cache in memory.

Example

The following example returns the cache size if it is enabled and issues a warning if it's disabled:

```
var cache = httpServer.cache;
if (cache.enabled) {
    cache.memorySize = newCacheSize;
} else {
    console.warn("cache disabled");
}
```

defaultCharSet

Description

The defaultCharSet property contains the current charset used by the Wakanda HTTP server for dynamic text responses by default. By defaut, the value is "UTF-8".

hostName

Description

The **hostName** property returns the host name of the server, if any. When you're running on a local machine, "localhost" is returned.

ipAddress

Description

The **ipAddress** property returns the IP address of the current application's server.

port

Description

The **port** property contains the TCP port number on which the server is listening for the current application. By default, when the server opens several applications, the port 8081 is used for the first application, 8082 for the second application, and so on.

ssl

Description

The **ssl** property is an object of type *HttpServerSSL* that contains two properties and a method:

- **enabled** - Boolean: The status of the SSL protocol on the server.
- **port** - Number: The port number for SSL connections. By default, port 443 is used.
- **getCertificatePath( )**: This method returns the path of the SSL certificate.

started

Description

The **started** property returns the current running status of the the HTTP server:

- **true**: the server has been started
- **false**: the server has not been started.

getCertificatePath( )

String **getCertificatePath**( )

Returns	String	Path to the SSL certificate

Description

The **getCertificatePath( )** method returns the path of the SSL certificate used by the server.

This method can only be applied to the **ssl** property of the HTTP server.

restart( )

void **restart**( )

Description

The **restart( )** method executes a sequence that stops and starts the Wakanda HTTP server. This method is useful when you need initialization parameters that were modified to be taken into account.

start( )

void **start**( )

Description

The **start( )** method starts the Wakanda HTTP server. If the server was already running, this method has no effect.

stop( )

void **stop**( )

Description

The **stop( )** method stops the Wakanda HTTP server. If the server was not running, the method has no effect.

Solution

The Solution class provides the interface for solution properties and allows administrators to manipulate all the application properties in a solution. You can access these properties and methods through the Global Application **solution** property.

applications

Description

The **applications** property returns an array containing the applications stored in the current solution. For more information about the Application objects, refer to the **Application** class description.

name

Description

The **name** property returns the name of the solution.

recentlyOpened

Description

The **recentlyOpened** property returns an array containing the names of all the recently opened solutions.

close( )

void **close**( )

Description

The **close( )** method closes the current solution and reopens the default solution.

Before closing the current solution, this method waits until all the code executed on the server has finished. It is therefore not recommended to write code that will be executed after this method.

getApplicationByName( )

Object **getApplicationByName** (name)

Parameter	Type	Description
name	String	Name of the application
Returns	Object	Application object

Description

The **getApplicationByName( )** method returns a reference to the application object whose name you passed in the *name* parameter. The application must belong to the current solution.

For more information about application objects, refer to the **Application** class description.

getDebuggerPort( )

Number **getDebuggerPort**( )

Returns	Number	Port number used for the debug service or -1 if the service in unavailable

Description

The **getDebuggerPort( )** method returns the port number on which Wakanda Server's debug service is listening for the *solution*.

By default, the Wakanda server debug service port number is 1919; however, this port is allocated dynamically. At server startup, if port 1919 is already used (for example, by another service or by another Wakanda Server), the server will try to open port 1920 ; if it is used, it will try to open port 1921, and so on, until a free port is found.

If the debug service is unavailable for any reason, this method returns -1.

Note: This method is also available through an RPCService.

getFolder( )

String | Folder **getFolder**([String *kind*] [, Boolean | String *encodeURL*])

Parameter	Type	Description
kind	String	Type in which the folder must be returned: path, URL, or Folder (default)
encodeURL	Boolean, String	True to encode the returned URL (default), otherwise false
Returns	Folder, String	Folder of the solution

Description

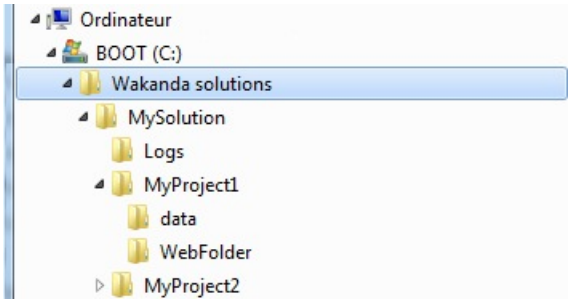
The **getFolder( )** method returns the folder containing the solution file (named '*SolutionName*.waSolution').

You can get the folder in different formats, depending on the string you pass in the *kind* parameter:

- If you pass "folder" or if you omit the *kind* parameter, the method returns an object of type *Folder*.
- If you pass "path", the method returns a string containing the path of the folder.
- If you pass "url", the method returns a string containing the URL of the folder.
  By default in this case, the URL is encoded. You can get this value without it being encoded by passing *false* to the *encodeURL* parameter.

Example

Considering the following organization of files and folders on disk:



The following example illustrates the different values that can be returned by **getFolder( )** when applied to a solution:

```
var theEncodedURL=solution.getFolder ("url");
    // returns file:///C:/Wakanda%20solutions/MySolution/
var theRawURL=solution.getFolder ("url", false);
    // returns file:///C:/Wakanda solutions/MySolution/
var thePath=solution.getFolder ("path");
     // returns C:/Wakanda solutions/MySolution/
var theFolder=solution.getFolder ("folder");
     // returns { name: "MySolution", extension: "", folders: Array(), 11 more}
```

getSettingFile( )

void **getSettingFile**(String *settingID* [, String *kind*] [, Boolean *format*])

Parameter	Type	Description
settingID	String	ID of the setting for which to get the file
kind	String	Type in which must be returned the file: path, relativePath, url, or file (default)
format	Boolean	True to encode the returned URL (default), otherwise false

Description

The **getSettingFile( )** method returns a reference or the path to the file containing the solution setting whose ID you passed in *settingID*. For example, if you want to get the *mySolution*.waSettings file containing the database settings, pass "database" in *settingID*.

The available setting IDs are the following:

- "solution": server startup options.
- "database": solution cache parameters.

By default, all the solution settings are gathered in a single settings file, named *solutionName*.waSettings -- any of the listed setting IDs will return the same information. But they could be stored in separate .waSettings files in the project, that's why the *settingID* parameter is designated.

You can get the file in different formats, depending on the string you pass in the *kind* parameter:

- If you pass "file" or if you omit the *kind* parameter, the method returns an object of type *File*.
- If you pass "path", the method returns a string containing the absolute path to the file.
- If you pass "relativePath", the method returns a string containing the relative path to the file.
- If you pass "url", the method returns a string containing the URL to the file.
  By default in this case, the URL is encoded. You can get this value without encoding by passing *false* to the *encodeURL* parameter.

getWalibFolder( )

String | Folder **getWalibFolder**([String *kind* [, Boolean | String *encodeURL*]])

Parameter	Type	Description
kind	String	Type in which the folder must be returned: path, URL, or folder (default)
encodeURL	Boolean, String	True to encode the returned URL (default), otherwise False
Returns	Folder, String	Wakanda Server walib folder path or reference

Description

The **getWalibFolder( )** method returns Wakanda Server's "walib" folder, which contains the libraries and services available client-side, such as WAF and RpcService.

Note that this method is available in the Solution and Application classes: both return the same result unless you pass "relativePath" to the *kind* parameter.

You can get the folder in different formats depending on the string you pass to the *kind* parameter:

- If you omit the *kind* parameter, this method returns an object of type *Folder* (see the **Folder Class** methods and properties).
- If you pass "path" or "relativePath", this method returns a string containing the path to the folder. If you pass "relativePath", the path will be relative to the application or solution folder depending on the object to which it is applied.
  By default, the path is expressed in posix syntax. You can get this value in the system's syntax by passing *false* or "system" in the *format* parameter (you can set the standard format back by passing *true* or "posix").

- If you pass "url", the method returns a string containing the folder's URL.
  By default, the URL is encoded. You can get this value without encoding by passing *false* or "notEncoded" in the *format* parameter (you can set the standard format back by passing *true* or "encoded" in this parameter).

Note: On Mac OS, posix and system paths are equivalent.

Example

The following example illustrates the different values that can be returned by **getWalibFolder( )**:

```
var theEncodedURL=solution.getWalibFolder("url");
    // returns file:///C:/Wakanda%20versions/Wakanda%20Server/walib/
var theRawURL=solution.getWalibFolder("url", false);
    // returns file:///C:/Wakanda versions/Wakanda Server/walib/
var thePath=solution.getWalibFolder("path");
    // returns C:/Wakanda versions/Wakanda Server/walib/
var theFolder=solution.getWalibFolder("folder");
    // returns { name: "walib", extension: "", folders: Array(), 11 more}...
```

open( )

void **open**(String *solutionPath*)

Parameter	Type	Description
solutionPath	String	Path to the solution to open

Description

The **open( )** method opens the solution designated by the *solutionPath* parameter. If a solution is already open, it will first be closed.

In the *solutionPath* parameter, pass the path to the solution by using "/" as separators.

Example

```
solution.open("C:/Wakanda projects/Families/Families.waSolution");
```

openRecent( )

void **openRecent**(String *solutionName*)

Parameter	Type	Description
solutionName	String	Name of a recently opened solution

Description

The **openRecent( )** method opens the solution whose name you pass in the *solutionName* parameter. The solution must have been previously opened and its name must be recorded in the *recentlyOpened* array, available as a property of the class (see **recentlyOpened**). If another solution is already open, it will be closed before *solutionName* is opened.

If the *solutionName* is not available or cannot be found in the *recentlyOpened* array, this method has no effect.

quitServer( )

void **quitServer**( )

Description

The **quitServer( )** method quits Wakanda Server. All currently opened Wakanda applications are closed beforehand.