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.
- 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. These actions are logged in the solution's log file and can be displayed in the Wakanda Studio Debugger's console.

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, ")");
```

This example could provoke the following lines to be added to the solution's log file:

```
2012—10—08 17:24:26 [com.wakanda-software.Contacts.Contacts] INFO - {"data":
["Update successful","3D Accounting for Windows"]}, HTTP connection handler
2012—10—08 17:24:26 [com.wakanda-software.Contacts.Contacts.console] INFO -
{"data":["Update successful","3D Accounting for Windows"]}, HTTP connection
handler
2012—10—08 17:24:26 [com.wakanda-software.Contacts.Contacts] ERROR -
{"data":["Update failed","3D Drawing for Windows","(Error:","Wrong
ID",")"]}, HTTP connection handler
2012—10—08 17:24:26 [com.wakanda-software.Contacts.Contacts.console] ERROR -
{"data":["Update failed","3D Drawing for Windows","(Error:","Wrong
ID",")"]}, HTTP connection handler
```

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 HttpServer 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.

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;
    }
);
```

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 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 **getItem( )** 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.

For more information about project settings, refer to the **Project Settings File** section in the *Wakanda Server Administration* manual.

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	Comment
project	object	publicName	""	
		listen	"0"	
		hostName	"localhost"	
		responseFormat	"json"	
		administrator	"false"	
database		object		
		journal.enabled	"true"	
		journal.journalFolder	"./DBjournal.journal"	
		autoRecovery.integrateJournal	"true"	
		autoRecovery.restoreFromLastBackup	"true"	
http	object	port	"8082"	
		allowSSL	"false"	
		SSLMandatory	"false"	

		SSLPort	"443"	
		SSLMinVersion	"3"	
		useCache	"false"	
		pageCacheSize	"5242880"	
		cachedObjectMaxSize	"524288 "	
		keepAliveMaxRequests	"100"	
		keepAliveTimeOut	"15"	
		acceptKeepAliveConnections	"true"	
		logFormat	"ELF"	
		logTokens	""	Added in v
		logFileName	"HTTPServer.waLog"	
		logPath	"Logs/"	
		logMaxSize	"10240"	
		allowCompression	"true"	
		compressionMinThreshold	"1024 "	
		compressionMaxThreshold	"10485760"	
webApp	object	-	-	Replaced b 'services' i v3
		documentRoot	"webFolder/"	Removed after v2
		directoryIndex	"index.html"	Deprecate after v2. U standard index.waP instead
dataService	object	-	-	Replaced b 'services' i v3
rpcService	object	-	-	Replaced b 'services' i v3
service	object	dataStore.name	"dataStore"	Added in v
		dataStore.modulePath	"services/dataStore"	
		dataStore.autoStart	"true"	
		dataStore.enabled	"true"	
		rpc.name	"rpc"	
		rpc.modulePath	"services/rpc"	
		rpc.proxyPattern	"^/rpc-proxy/"	
		rpc.autoStart	"true"	
		rpc.enabled	"true"	
		rpc.publishInClientGlobalNameSpace	"false"	Deprecate in v3
		webApp.name	"webApp"	
		webApp.modulePath	"services/webApp"	
		webApp.autoStart	"true"	
		webApp.enabled	"true"	
		upload.name	"upload"	Added in v

		upload.modulePath	"services/upload"	Added in v
		upload.autoStart	"true"	Added in v
		upload.enabled	"true"	Added in v
		upload.maxSize	""	Added in v
		upload.maxFiles	""	Added in v
		upload.sizeUnity	"mb"	Added in v
		remoteFileExplorer.name	"remoteFileExplorer"	Added in v
		remoteFileExplorer.modulePath	"services/remoteFileExplorer"	Added in v
		remoteFileExplorer.autoStart	"true"	Added in v
		remoteFileExplorer.enabled	"true"	Added in v
		Git HTTP Service.name	"Git HTTP Service"	Added in v
		Git HTTP Service.modulePath	"services/waf-git/waf-GitService"	Added in v
		Git HTTP Service.autoStart	"true"	Added in v
		Git HTTP Service.enabled	"true"	Added in v
resources	object	location	"/walib/"	
		lifeTime	"31536000"	
javaScript	object	reuseContexts	"true"	

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	String	Wakanda internal build version, for example "2.108407"
version	String	Wakanda version full string, for example "2.0 build 2.108407"

wildchar

Description

The **wildchar** property returns the wildcard character defined for Wakanda queries, which is the asterisk (*). As defined in the Defining Queries (Server-side) section, the wildcard character can be used to define queries of the type "starts with" or "contains".

You can use this property into your query parameters to make your code more readable.

For example, these queries:

```
   // find employees with first name begins with "John"
var coll = ds.Employee.query( "firstName = :1", "John*");
    // find employees with job contains "dev"
var coll2 = ds.Employee.query( "jobName= :1", "*dev*");
```

can also be written:

```
var coll = ds.Employee.query( "firstName = :1", "John"+wildchar);
var coll2 = ds.Employee.query( "jobName= :1", wildchar+"dev"+wildchar);
```

***Note:** The* WAF.wildchar *property is available client-side.*

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 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*, object of the HTTPRequest type
    - *response*, object of the HTTPResponse type

  The function also can return a value by using the **return** statement.

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

Defining the pattern Parameter

The *pattern* parameter sets the requests to be intercepted and processed using the HTTP request handler.

To define this parameter, you must use a Regex (Regular expression). Here are a few principles for pattern definitions that are generally used in Web applications:

- **^/myPattern$**: intercepts requests containing only the "/myPattern" pattern, for example *"GET http://mydomain.com/myPattern"*.
  "/files/myPattern" or "/myPattern/myvalue" type requests are not taken into account.
- **^/myPattern/**: intercepts requests that begin with the "/myPattern/" pattern. This pattern intercepts any requests beginning with "/myPattern/myfoldr" as well as requests like "/myPattern/myfile.html" or "/myPattern/myfile.js".
- **^/myPattern[?]***: intercepts requests that begin with the "/myPattern" pattern and that contain, optionally, a query string. You can use this pattern to handle requests containing "/myPattern?Name="Martin", and so on.
- **/myPattern**: intercepts any requests containing /myPattern regardless of its position in the string. This pattern will accept indifferently requests such as "/files/myPattern", "/myPattern.html", and "/myPattern/bar.js".

Wakanda server supports ICU's Regular Expressions package. For more information, please refer to the following address:
http://www.icu-project.org/userguide/regexp.html

For more information about Regex in general, please refer to the following address:
http://en.wikipedia.org/wiki/Regular_expression

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 (or "/DoGetStuff", "/dogetstuff", etc.).

addRemoteStore( )

Datastore **addRemoteStore**(String *storeName*, Object *params*)

Parameter	Type	Description

storeName	String	Name of the datastore reference in global object
params	Object	Parameters for the remote datastore access
Returns	Datastore	Reference to the remote datastore

Description

The **addRemoteStore( )** method adds the remote datastore defined by *params* to the current project and maintains a global reference to it.

Unlike the openRemoteStore( ) method, **addRemoteStore( )** adds and maintains a dynamic reference to the remote datastore in the global object. Thus, it will be available in all JavaScript contexts during the whole session. It can be called in the bootstrap file, for example. The connection will be kept alive until the project is closed.
In *storeName*, pass a string representing the global property name whose value will be the remote *Datastore*. For example, if you pass "my4DBase" in *storeName*, you can then write "my4DBase.Invoices.all()". You can use any valid name that does not belong to the Wakanda Reserved Keywords list.

The remote target datastore can be one of the following servers:

- another Wakanda Server
- a 4D Server (v14 and higher)
- an SQL database

In the *params* object, pass a set of properties that defines the remote datastore to which you want to connect. The required properties depend on the kind of datastore you want to open:

Property	**Value type**	**Description**	**Wakanda Server**	**4D Server**	**SQL database**
hostname	*String*	Full name or IP address of the remote server	X	X	X
ssl	*Boolean*	Enable SSL connection	X	X	X
user	*String*	User name	X	X	X
password	*String*	User password	X	X	X
SQL	*Boolean*	**true** to connect to an SQL server			X
port	*Number*	Port number			X
database	*String*	Database name			X

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 do with the ds object.

Since connection to the remote datastore is synchronous, the *storeName* reference can be used in your code just after the **addRemoteStore( )** method call.

Example

You want to reference an external datastore permanently in your project:

```
var extInfo =
    {   hostname: "www.mylib.com",
        ssl: true,
        user: "john",
        password: "pass"
    };
var extStore = addRemoteStore("myLibrary", extInfo);
    // synchronous connection, so you can write
var coll = myLibrary.Books.all();
```

backupDataStore( )

Null **backupDataStore**(File *model*, File *data*, Object *settings*[, Object *options*])

Parameter	**Type**	**Description**
model	File	Datastore's model file to back up
data	File	Data file to back up
settings	Object	Backup settings
options	Object	Block of callback functions
Returns	Null	Backup manifest

Description

The **backupDataStore( )** method starts the backup of the closed datastore defined by *model* and *data*.

***Note**: The **backupDataStore( )** method has the same functioning as the ds.**backup( )** method, except that it only applies to a closed datastore. Also, unlike the ds.**backup( )** method, you have to specify all the backup settings because default parameters cannot be accessed.*

As explained in the **Backup and Restore Overview** section, a backup operation will archive the data folder and the journal file (if activated). If the datastore was using a journal, that journal is reset.

- *model*: pass a reference to the datastore's model file (whose extension is ".waModel").
- *data*: specify a reference to the *model*'s data file (whose extension is ".waData") to back up.

You cannot designate files that are currently open -- the datastore must be closed. If you attempt to use this method to back up a datastore that is currently open, an error will be generated.
The data file designated is opened in read-only mode. Before calling this method, make sure that no application has access to this file in write mode since otherwise the backup operation could fail.

Example

The following function shows an example of how to call the **backupDataStore( )** method:

```
function demoBackup()
{
    function myOpenProgress(title, maxElements)
     {
         console.log(title+" on "+maxElements+" operations"); // log the proc
     }

    function myProgress(curOp, maxElements)
    {
        console.log("current element: "+curOp);  // log each operation
            // events can be nested
    }

    function myCloseProgress()
    {
        ... // add code to handle the closing of the progress event
    }

    function myAddProblem(problem)
    {
        if (problem.ErrorLevel <= 2) // we only handle fatal or regular erroı
        {
            this.storedProblems.push(problem);  // fill the custom array
            console.log(problem.ErrorText);  // log the error description
        }
    }

    var modelFile = File("c:/wakanda/mySolution/people.waModel");
    var modelData = File("c:/wakanda/mySolution/people.waData");
```

```
        var settings = {
            destination:Folder("c:/backups/"),
            useUniqueNames:true,
            backupRegistryFolder: Folder("c:/backups/Regs/")
            };


        var options = {
            'openProgress': myOpenProgress,
            'closeProgress': myCloseProgress,
            'progress': myProgress,
            'addProblem': myAddProblem,

                // you can add any custom code here, it will be passed to the
                // addProblem function in the 'this' keyword
            'storedProblems':[]   // we add an array to store any problems that a
            };

        var manifest = backupDataStore(modelFile, modelData, settings, options);

        if (options.storedProblems.length > 0)
        {
            ... // code to warn the user that some problems have occurred
        }
        else
        {
            ... // everything is ok
        }
    }
```

BinaryStream( )

BinaryStream **BinaryStream**(String | File | SocketSync | Socket *binary* [, String *readMode*] [, Number *timeOut*])

Parameter	Type	Description
binary	String, File, SocketSync, Socket	Binary file or socket to reference
readMode	String	Streaming action: "Write" to write data, "Read" to read data
timeOut	Number	Timeout in milliseconds (used for socket objects only)
Returns	BinaryStream	New BinaryStream object

Description

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

In the *binary* parameter, pass the path of, or a reference to, the binary file or socket to write or to read. It can be one of the following forms:

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

- a *Socket* or *SocketSync* object. For more information on this objects, refer to the section.
  If you passed a socket as *binary* parameter, you can define the *timeOut* parameter.

Once the file is referenced, you can start writing or reading the 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" or omit the *readMode* parameter, the file is opened in read mode.

The *timeOut* parameter is useful when you work with sockets. It allows you to define the timeout in milliseconds when reading or writing data in the socket. If this parameter is omitted, there is no default timeout (the method waits indefinitely).

Example

This example shows how to handle a socket as a BinaryStream:

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

var socket = net.connectSync(25, "smtp.gmail.com");
var readstream = BinaryStream(socket, "Read", 300);
var writestream = BinaryStream(socket, "Write", 500);

console.log(readstream.getBuffer(1000).toString())

try {
        // Server expects us to send EHLO or HELO command,
        // so there is nothing to read.
    readstream.getByte();
} catch (e) {
    console.log("OK timedout\n");
}

writestream.putBuffer(new Buffer("EHLO\r\n"));
    // Read answer from the server for EHLO command

console.log(readstream.getBuffer(1000).toString())
```

Blob( )

void **Blob**(Number *size* [, Number *filler*] [, String *mimeType*])

Parameter	Type	Description
size	Number	Size of the new BLOB in bytes
filler	Number	Filler character code value
mimeType	String	Media type of the BLOB

Description

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

Pass in *size* the expected size of the *Blob* in memory. The size must be expressed in bytes.

If you want to initialize each byte of the *Blob* to a specific character, pass the corresponding character code into the *filler* optional parameter. For example, pass 88 to fill the BLOB with "X". By default if you omit this parameter, the *Blob* is filled with "0" (zeros).

In the optional *mimeType* parameter, you can pass a lower case string representing the media type of the Blob, expressed as a MIME type (see RFC2046). By default if you omit this parameter, the *Blob* media type is "application/octet-stream".

Example

We want to create a 20 bytes Blob, filled with X and associated to the standard binary MIME type:

```
var myBlob = new Blob( 20 , 88, "application/octet-stream");
var myString = myBlob.toString();
//myString contains "XXXXXXXXXXXXXXXXXXXX"
```

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).
  - 'hex' - Encode each byte as two hexadecimal characters.
  - 'base64' - Base64 string encoding.
    Note: *A specific processing is applied when you pass 'base64' in encoding: with this statement, you indicate that the string is already encoded in 'base64'. Thus, the constructor will only create a buffer containing the base64-encoded string binary data. No encoding is done actually. Of course, the string must be a valid base-64 encoded string.*

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 (their internal "close" flag is set to **true**). If **close( )** is called during a callback in a **wait( )**, this will exit the **wait( )**.
- From a child thread.
  In this case, the internal "close" flag is set to **true**. The **wait( )** event loop is exited and the thread is closed.

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

compactDataStore( )

void **compactDataStore**(File *model*, File *data*[, Object *options*], File *compactedData*)

Parameter	Type	Description
model	File	Datastore's model file to compact
data	File	Data file to compact
options	Object	Callback functions
compactedData	File	Destination for the compacted data file

Description

The **compactDataStore( )** method compacts the datastore's data file designated by *model* and *data*, and generates the *compactedData* data file.

- *model*: pass a reference to the datastore's model file (whose extension is ".waModel").
- *data*: specify a reference to *model*'s data file (whose extension is ".waData") to compact.
- *compactedData*: pass a reference to the compacted data file (whose extension must be ".waData") that will be created.

You can designate the current *model* file, but not the current *data* file. If you attempt to verify a data file that is currently open, an error will be generated. The *data* data file is opened in read only mode before it is compacted.

Make sure that the path for *compactedData* is valid and that you have enough space on your destination disk for the duplicated data file.

Compacting data is useful because it reorganizes and optimizes the data stored in the data file by removing the unused spaces (“holes”). When you delete entities, datastore classes, and so on, the space that they previously occupied in the data file becomes empty. Wakanda tries to reuse these empty spaces whenever possible, but since the size of data is variable, successive deletions and/or modifications will inevitably generate spaces that cannot be reused. The same is true when a large quantity of data has just been deleted: the empty spaces remain unused in the data file.

If the method encounters a level 2 error (standard error) or a level 1 error (fatal error), it stops executing. For more information about error levels, see the *options* parameter description.

options

In the *options* parameter, you can pass an object containing one or more callback functions. These functions will be called automatically by Wakanda's maintenance or backup engine (with the relevant parameter(s)) when the corresponding event occurs. You then need to write code to handle the information returned by these functions.

Note: *You can omit the options parameter when processing a data file, but if an error occurs, you will not know the reason why it occurred.*

The *options* parameter has the following callback functions:

- **openProgress: function(title, maxElements)**
  This function is called each time the engine starts processing a new group of elements or a new file, i.e., for each new **progress event**. For a maintenance operation, the event could be the processing of entities for a datastore class, an index table, etc. For a backup operation, it could be the backup of the journal file, the backup initialisation or the backup of the data file. This function is called with the following parameters:
    - *title* (string): label for the new progress event (e.g., "Verifying entities in datastore class #3").
    - *maxElements* (number): total number of elements to handle in the group (e.g., the number of entities in the datastore class).

  You can use this function, for example, to open a progress indicator on the client to display the progress or to write the progress of the operation to a log file on disk.

- **closeProgress: function()**
  This function is called when a progress event is closed. Progress events can be nested at multiple levels (e.g., one progress event can start checking all the indexes and another one opens for each index). A closeProgress event is generated for each openProgress event.

- **progress: function(curElement, maxElements)**
  This function is called during a progress event for each element to be verified, repaired, compacted (e.g., an entity or an index page) or backed up. This function is called with the following parameters:
    - *curElement* (number): position of the element currently being handled in the group.
    - *maxElements* (number): total number of elements to handle in the group (same value as in openProgress).

- **setProgressTitle: function(title)**
  This function is called when the title of a progress event has changed for some reason (e.g., when a missing piece of information is found while verifying the data). This function is called with the following parameter:
    - *title* (string): new label for the progress event.

Note: *This function is rarely called.*

- **addProblem: function(problem)**
  This function is called when a problem has been detected in the data during a progress event. If an error occurs, only this function is called. It is therefore mandatory to call this function if you want to know whether the verification of a data file was successful or not. This function is called with the following parameter:
  - *problem* (object): object describing the problem. Information provided in this object depends on the type of the problem. For example, if an entity cannot be read, the entity number is returned in the *ErrorText* property.
    The *problem* object is returned with the following properties:
    - *problem.ErrorText* (string): complete description of the error
    - *problem.ErrorNumber* (number): error number
    - *problem.ErrorType* (number): internal code that returns the type of the error
    - *problem.ErrorLevel* (number): level of seriousness of the error. Available values are:
      - 1=fatal error (currently unused)
      - 2=regular error: serious errors including various kind of issues (wrong index type, invalid file header, checksum errors, data and model inconsistencies...). Some regular errors may require the datastore to be repaired.
      - 3=warning: minor errors, such as indexes that need to be rebuilt or externally stored blob field content that is not found.

Example

The following function shows an example of how to call the **compactDataStore( )** method:

```
function demoCompact()
{
    function myOpenProgress(title, maxElements)
     {
         console.log(title+" on "+maxElements+" elements"); // log the progre
     }

    function myProgress(curElement, maxElements)
    {
        console.log("current element: "+curElement);  // log each element
            // events can be nested
    }

    function myCloseProgress()
    {
        ... // add code to handle the closing of the progress event
    }

    function myAddProblem(problem)
    {
        if (problem.ErrorLevel <= 2) // we only handle fatal or regular erro
        {
            this.storedProblems.push(problem);  // fill the custom array
            console.log(problem.ErrorText);  // log the error description
        }
    }

    var modelFile = File("c:/wakanda/mySolution/people.waModel");
    var modelData = File("c:/wakanda/mySolution/people.waData");
    var compactDest = File("c:/wakanda/mySolution/compactedData.waData");

    var options = {
        'openProgress': myOpenProgress,
        'closeProgress': myCloseProgress,
        'progress': myProgress,
        'addProblem': myAddProblem,

            // you can add any custom code here, it will be passed to the
            // addProblem function in the 'this' keyword
        'storedProblems':[]   // we add an array to store any problems that a
        }

    compactDataStore(modelFile, modelData, options, compactDest);

    if (options.storedProblems.length > 0)
    {
        ... // code to warn the user that some problems have occurred
    }
    else
    {
        ... // everything is ok
    }
}
```

currentSession( )

ConnectionSession **currentSession**( )

Returns	ConnectionSession	Object containing the current user session properties

Description

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 logged user session, the default user session is returned ("default guest").

Example

To get the user running the current session on the server:

```
var curSession = currentSession();
var curUser = curSession.user;
```

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 logged user session, the default user is returned ("default guest").

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;
}
```

DataStoreCatalog( )

void **DataStoreCatalog**( )

Description

The **DataStoreCatalog( )** method is the constructor of the *Model* type objects. It allows you to create a model procedurally (also called a datastore catalog) for your Wakanda application. Model objects are handled using the Model class.

To activate the procedural model mode and create the *model* object, you must:

- call this method in the "Model.js" file located at the root of your Wakanda project,
- reference a new global object named "model",
- use the **new** operator to create an instance of the object.

Once your model is instantiated with this method, it is automatically loaded at Wakanda server launch and its contents, including datastore class methods, are available in all JavaScript contexts. Note also that the model can also be loaded in Wakanda Studio's Model Designer.

***Note**: You can initialize a JavaScript* model *object in your Wakanda application by creating a standard JavaScript global object named "**model**" in the Model.js file, for example:*

```
model = {}; //creates a dynamic model
```

*This code activates the procedural model mode. However, when using the **DataStoreCatalog( )** constructor, you create a prototyped Model object and benefit from the various API methods of the Model class, such as **addClass( )**.*

Example

You want to initialize a model in your project and call a model definition that is defined in other external files. You write the following code in the *Model.js* file at the root of your project:

```
model = new DataStoreCatalog(); //initializes model instance in the project
include("classes/emp.js"); //file containing Employee class description
include("classes/comp.js"); //file containing Company class description
```

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**(*projectName* [, *projectName*] [, Boolean *critical*])

Parameter	Type	Description
		Message to display
		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).

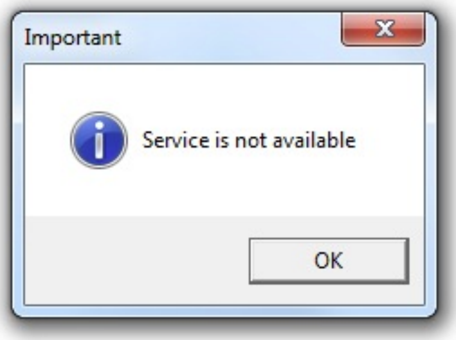
An alert dialog is shown on Mac OS and Windows. On Windows, an "info" icon is shown by default. If you set the *critical* parameter to *true*, a "warning" icon is used.

Example

When you execute the following code:

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

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



Service is not available

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 *absolutePath*)

Parameter	Type	Description
absolutePath	String	Full pathname of the file to reference
Returns	File	New File object

Description

The **File( )** method is the constructor of the *File* type objects. *File* objects are handled using the various properties and methods of the **File**.

With the first syntax, you only pass in *absolutePath* an absolute file path using the POSIX syntax or a URL, including the file name.

With the second syntax, you can pass an object referencing a *path* as first parameter and a relative file path in the *fileName* parameter. You can pass in the *path* parameter:

- a *Folder* object -- in this case, the *fileName* parameter contains a path relative to the *Folder* object, including the file name
- a *FileSystemSync* object -- in this case, the *fileName* parameter contains a path relative to the *FileSystemSync* root folder, including the file name

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 ref
    var myDS = createDataStore(currentModel, dataFile);     // create and ref
}
```

FileSystemSync( )

FileSystemSync **FileSystemSync**(Number *type*, Number *size*)

Parameter	Type	Description
type	Number	0 = Temporary, 1 = Persistent
size	Number	Expected storage space (in bytes)
Returns	FileSystemSync	New synchronous file system object

Description

The **FileSystemSync( )** method requests a *FileSystemSync* object in which application data can be stored. If the method is executed successfully, a new *FileSystemSync* object is returned.

Pass in *type* a value describing the type of storage space you want the file system to reserve. You can pass either a number value or a constant:

Constant	Value	Description
TEMPORARY	0	Used for storage with no guarantee of persistence. Data stored in a temporary storage may be deleted at the server's convenience, e.g. to deal with a shortage of disk space. In Wakanda, temporary storage is deleted after the Javascript context is closed.
PERSISTENT	1	Used for storage that should not be removed by the server without user permission. Data stored there will not deleted after the Javascript context is closed.

Pass in *size* a value indicating how much storage space, in bytes, the server expects to need.

Example

Create a temporary file system:

```
var myFileS = FileSystemSync ( TEMPORARY, 1024*1024);
```

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**.

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"
```

getBackupRegistry( )

Array **getBackupRegistry**(Folder *registryFolder*)

Parameter	**Type**	**Description**
registryFolder	Folder	Folder containing the registry to read
Returns	Array	Array of the most recent backup manifests (up to 20)

Description

The **getBackupRegistry( )** method returns an array that lists the 20 most recent backup manifests recorded in the specified backup registry. A backup registry (named *lastBackups.json*) contains a list of the last backup manifests.

***Note**: **getBackupRegistry( )** is useful when you defined a custom backup registry folder while executing your backups. If you only want to get the last manifests from an application that uses default settings, you might consider using **getLastBackups( )** instead.*

In *registryFolder*, pass the *Folder* containing the you want to read. You can pass the registry folder for any application of the current solution. The backup registry folder path can be set either by using default settings (see) or when using **backup( )** or **backupDataStore( )** methods. The path can be obtained by querying an application's backup settings.

In the registry as well as in the returned array, backup manifests are ordered from the oldest (at the top) to the most recent entries (at the bottom).
A "path" property is added at the beginning of each element of the array; it gives the full path (POSIX syntax) of the corresponding manifest file.

Example

You want to get manifests extracted from the registry file located in the "MyBackupRegistryFolder" subfolder:

```
    // set a custom folder for the registry
var settings = getBackupSettings();
```

```
settings.backupRegistryFolder = new Folder("/home/dbuser/MyBackupRegistryFol
    // do a backup
var manifest = ds.backup(settings);
    //Backup is executed using default application settings, however
    //the registry folder is not the default one

    //This call will return the content of the registry you specified for the
var backupManifests = getBackupRegistry(settings.backupRegistryFolder);
```

getBackupSettings( )

Object **getBackupSettings**( )

Returns	Object	Default backup settings

Description

The **getBackupSettings( )** method returns an *Object* containing the default backup settings for the solution. For more information about these settings, refer to the Backup Settings paragraph.

Default backup settings cannot be modified but you can edit the returned object and pass it as a parameter to the backup( ) or backupDataStore( ) method.

Example

You want to back up the current datastore with a custom setting:

```
var mySettings = getBackupSettings(); //get the settings object
    //edit one property
mySettings.backupRegistryFolder = new Folder("/home/dbuser/MyBackupRegistryFo
    // back up with modified settings
ds.backup(mySettings);
```

getDataStore( )

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

Parameter	Type	Description
projectName	String	Solution project name for which you want to get the current datastore
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
var mySet = dsTemp.Clients.query(country = "France");      // perform a search
```

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 th
  ```

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

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

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 .waProject extension).

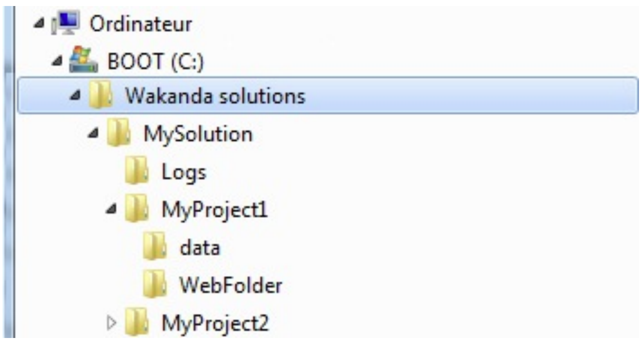
You can get the folder in different formats, depending on the string you pass in the *kind* parameter:

- If you omit the *kind* parameter, the method returns an object of the *Folder* type (see the Folder 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/
```

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). See Project Settings File.
"catalog"	File	Model file of the application (.waModel). See Projects.
"bootStrap"	File or array of Files	JavaScript file(s) automatically executed at the project launch (can contain HTTP request handlers). See Using Bootstrap File.
"webFolder"	Folder	Folder containing all the files that Wakanda will send to the Web. See Projects.
"dataFolder"	Folder	Application's active data folder (named "DataFolder" by default). See Projects.
"backup"	Folder	Application's default backup folder.

getLastBackups( )

Array **getLastBackups**( )

Returns	Array	Array of the most recent backup manifests (up to 20)

Description

The **getLastBackups( )** method returns an array that lists the 20 most recent backup manifests recorded in the backup registry default folder of the application. A backup registry (named *lastBackups.json*) contains a list of the last backup manifests.

The backup registry default folder path is set in the default settings (see). This path can be obtained by querying an application's backup settings.
If you defined another folder by using backup( ) or backupDataStore( ) methods, you need to use the getBackupRegistry( ) method instead.

In the registry as well as in the returned array, backup manifests are ordered from the oldest (at the top) to the most recent entries (at the bottom).
A "path" property is added at the beginning of each element of the array; it gives the full path (POSIX

syntax) of the corresponding manifest file.

Example

Get the contents of the last backup manifests for the current application:

```
var manifests = getLastBackups();
```

Get the contents of the last backup manifests for the "Cities" application - which is running but is not the current application:

```
var myApplication = getApplicationByName("Cities");
var manifests = myApplication.getLastBackups();
```

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** 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.waSett
```

getURLPath( )

Array **getURLPath**(*projectName*)

Parameter	Type	Description
		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**(*projectName*)

Parameter	Type	Description
		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/?$top=40&$me
        // theQuery is {$top:40, $method:entityset, $timeout:300}
        // theQuery.$top is 40
```

getUserSessions( )

Array **getUserSessions**([User | String *user*])

Parameter	Type	Description
user	User, String	User, user name or user ID
Returns	Array	Running user session(s)

Description

The **getUserSessions( )** method returns an array of user sessions running on the server for the current application. Each element of the returned array is a *connectionSession* object. User sessions can be handled by the Session properties and methods.

If you omit the *user* parameter, the array will contain all the running user sessions, including default guest sessions.

Otherwise, you can pass a *user* parameter to get only the sessions for a specific user by specifying one of the following:

- the name property of a user,
- the ID property of a user, or
- a User object.

Example

You want all sessions of the current user to expire:

```
var arrStop = getUserSessions(currentSession().user);
arrStop.forEach(function(item) {
    item.forceExpire();
});
```

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 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%20versi
var theRawURL=getWalibFolder("url", false); // returns file:///C:/Wakanda vers
var thePath=getWalibFolder("path"); // returns C:/Wakanda versions/Wakanda Se
var theFolder=getWalibFolder(); // returns { name: "walib", extension: "", fc
```

importScripts( )

void **importScripts**(String *scriptPath*)

Parameter	Type	Description
scriptPath	String	Pathname(s) to JavaScript file(s)

Description

The **importScripts( )** method allows you to import and execute any JavaScript file(s) in the current JavaScript context.

In the *scriptPath* parameter, pass one or several path(s) to JavaScript file(s). If you use a relative path, Wakanda assumes that the project folder is the default folder. The referenced file(s) must have valid JavaScript statements, otherwise an error is generated.

You can pass several files in *scriptPath*, separated by commas. **importScripts( )** will execute all the queued files and return the result from the last evaluated script of the list. If an error occurs during the execution of a script, an error is generated and the remaining files are not executed.

***Note:** The Wakanda **importScripts( )** method is compliant with the Web Workers W3C specification.*

Example

Imports and executes two scripts located in a "scripts" folder of the project folder:

```
var res = importScripts( // res will be set to the result of compute.js
    "scripts/init.js",
    "scripts/compute.js");
```

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!");
```

integrateDataStoreJournal( )

void **integrateDataStoreJournal**(File *model*, File *data*, File *journal*[, Object *options*])

Parameter	Type	Description
model	File	Datastore's model file (must not be open)
data	File	Data file (must not be open)
journal	File	Journal file to integrate
options	Object	Integration options

Description

The **integrateDataStoreJournal( )** method allows you to partially or fully integrate a *journal* file into a datastore. Both the *model* and *data* files must not be open nor accessed during the operation, otherwise it would fail.

Pass as parameters File objects containing valid references to the files involved in the integration operation:

- *model*: pass a *File* reference to the datastore's model file (whose extension is ".waModel").
- *data*: specify a *File* reference to the *model*'s data file (whose extension is ".waData") where you want to integrate the journal.

- *journal*: pass a *File* reference to the journal file to integrate. The *journal* file must match the *data* file. As explained in the paragraph, the integration of a log file can only be carried out in the database to which it corresponds. Usually, if you have *n* backups (where *Bn* is the most recent one), you can integrate *Bn*'s journal into *B(n-1)*'s data file.

By default, if you omit the *options* parameter, **integrateDataStoreJournal( )** will integrate all operations that are recorded in the *journal* file and the original *data* file will be directly updated. You can modify this default behavior by passing extra properties in the *options* parameter (object). The following properties are available:

- **fromOperation**: *number* (example: **fromOperation: 60**)
  Number of the first operation to integrate. By default, the value is 0: integration starts at the first operation.

- **upToOperation**: *number* (example: **upToOperation: 1000**)
  Number of the last operation to integrate. By default, the value corresponds to the last operation recorded in the journal (all operations are integrated).
  Note that since operations are uniquely identified within the journal, only a contiguous subset of operations can be integrated.

- **resultDataFile**: *file* (example: **resultDataFile: File("/var/data/restoredData.waData"**))
  Data file that will store the integrated operations. When you pass this property, the *data* file is duplicated at the specified location and the integration is performed on the copy. The original *data* file is left untouched.
  Warnings:
    - Any *File* specified in **resultDataFile** is silently overwritten (except if it is the same file as *data*).
    - Failing to copy *data* to **resultDataFile** results in an aborted integration.

Example

This example will integrate all operations from the journal into the standard data file (default):

```
integrateDataStore(
    File("/var/MyDb/model.waModel"),
    File("/var/MyDb/DataFolder/data.waData"),
    File("/var/MyDb/DataFolder/journal.waJournal")
);
```

Example

In this example, integration will start at operation 1024 and continue to the last operation:

```
var options = {fromOperation: 1024};
integrateDataStore(
    File("/var/MyDb/model.waModel"),
    File("/var/MyDb/DataFolder/data.waData"),
    File("/var/MyDb/DataFolder/journal.waJournal"),
    options
);
```

Example

In this example, integration will cover operations from the beginning of the journal up to operation 2054, and the regular data file will be left untouched:

```
var options = {
    upToOperation: 2054,
    resultDataFile: File("/var/MyDb/IntegratedData.waData")
};
```

```
integrateDataStore(
    File("/var/MyDb/model.waModel"),
    File("/var/MyDb/DataFolder/data.waData"),
    File("/var/MyDb/DataFolder/journal.waJournal"),
    options
);
```

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 str
    xmlText = JSONToXml(jsonText, "json-bag", root);   // convert the JSON in
    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 ob
"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/rest/Emplc
                      Name="Smith" jobName="Webmaster" salε
</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: *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*.

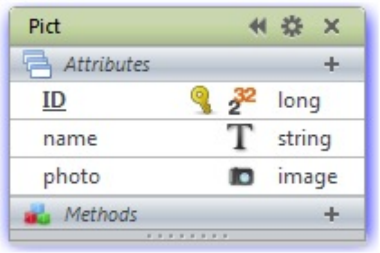
For more information about Wakanda image object manipulation, refer to the Images class description.

Note for Linux Users: *In the current version of Wakanda, the Image API is not supported on Linux platforms.*

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:



Pict		
Attributes		
ID		long
name		string
photo		image
Methods		

```
var mypict = loadImage ("C:/Wakanda/Solutions/mysolution/Tulips.jpg"); // loa
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	Computed key associated to the user
timeOut	Number	User session timeout (in seconds)
Returns	Boolean	True if the user has been successfully logged, otherwise False

Description

The **loginByKey( )** method authenticates a user by their *name* and *key* and, in case of success, opens a new user Session on the server.

The login request is accepted:

- When the user *name* and *key* are registered in the Directory of the application (for more information, please refer to the section Users and Groups) or
- (starting with v5) When the user *name* and *key* are processed successfully in your custom *LoginListener* function installed by the setLoginListener( ) method.

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 computed key value of the user you want to log in. Usually, this key will result from a hash computation, for example a SHA-1 computation combining the user's password and other infiormation, but actually you can use any value you want, resulting from any custom function. The same computation must have been done on the client side, so that the received key and the server stored key can be compared using the *LoginListener* function. Using a key challenge is more secure because it avoids sending the password itself over the network.

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).

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).

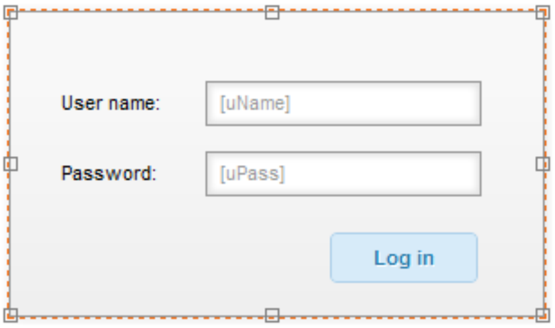
Example

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 cla
};
```

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 functio
{
   var result = loginByPassword(userName, password, 60*60); // session i
   return result; // result is sent to the client
}
```

logout( )

Boolean **logout**( )

Returns	Boolean	true if the user has been successfully logged out

Description

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**.

Mutex( )

Storage **Mutex**(String *key*)

Parameter	**Type**	**Description**
key	String	Mutex key
Returns	Storage	New mutex object

Description

The **Mutex( )** method creates a new mutex object that will allow you to control multithreaded concurrent accesses to JavaScript code. For example, you may want to protect a variable value from being modified by a thread B while they it is edited in a thread A. Mutex means *MUTual EXclusion* and designates a lock that can be open, or closed.

Pass a string in the *key* parameter. When you call **Mutex( )**, this parameter will act as a the key for mutex object. Any other thread that uses the same key will interact with the same mutex.

A mutex object has three methods:

- lock( )
- unlock( )
- tryToLock( ): this function attempts to lock the mutex and return true if it could be locked, and false if it couldn't.

Example

Here is a typical structure for a mutex controlled code:

```
var writeMutex = Mutex('writeMutex');
if (writeMutex.tryToLock())
    {
        //code here executes if tryToLock was able to lock the mutex
        writeMutex.unlock();
    }
    //code here executes even if tryToLock was NOT able to lock the mutex
```

Example

A mutex provides a way to pause execution in one thread until a condition is met in another. In the example below, we want to create a re-usable log file object for use in various server-side functions. To do this, we create a new JavaScript file (named myLog.js in our example) with the following code:

```
function Log(file)     //will use this as the constructor
 {
    this.logFile = null;
    this.init(file);
    return this;
 }

 Log.prototype.init = function(file) //add to the prototype chain of Log
```

```
{
    if(typeof file == 'string')     //if we pass a string
        file = File(file);      //assume it to be a path
    this.logFile = file;      //otherwise assume it to be a file
    if (!file.exists)     //if it doesn't exist
        file.create();     //create the empty file
}

 Log.prototype.append = function(message)     //add another function to Log
{
    if(this.logFile != null){
        //if the Log references a valid file, get a Mutex
        //using the files path as its name
        var logMutex = Mutex(this.logFile.path);
        //attempt to lock the Mutex
        //if it is already locked, the next line pauses execution
        //until it becomes unlocked
        logMutex.lock();
        //from here until logMutex.unlock() we know only one thread
        //can be writing to the log file
        var logStream = TextStream(this.logFile,"write");
        var today = new Date();
        var stamp = today.toDateString() + " " + today.toLocaleTimeString();
        logStream.write(stamp + ': ' + message +"\n");
        logStream.close();
        logMutex.unlock();
    }
}
```

The code above creates a new Log object and then adds two functions to its prototype chain. The first function named init takes either a Wakanda file object or a path to a file as a string. The code then checks that the file exists and if it doesn't, creates it. The next function, named append, takes a string as a parameter. The append function double-checks that the Log object has been initialized (i.e., references a valid log file) and then creates an object by calling **Mutex( )**. In our example we use the log file's path as our key with the goal that only one thread can write to a given file at the same time. We then include the above JavaScript file in our project's code by adding an include statement to our project's main JavaScript file (i.e., projectname.waModel.js) like this:

```
include('myLog.js');
```

This project's main JavaScript file is executed for all calls involving the project so the Log object will be available throughout.

To see an example of the Log object in action we will employ the verify( ) method, which verifies the data and indices of a Wakanda datastore. Consider the following class method:

```
function verifyData()
{
    var projectLog = new Log('./log.txt');
    //our projects main log file
    var noProblems = true;     //flag to let us know if there were any probler
    //the object below will provide the call back function(s)
    var problemHandler = {
        addProblem: function(problem){
            projectLog.append('Verify: ' + JSON.stringify(problem));
            noProblems = false;
        }
    }
     projectLog.append('Start Verify');
    ds.verify(problemHandler);     //will make periodic call backs
    if (noProblems)
        projectLog.append('Verify found no problems');
    projectLog.append('End Verify');
    return noProblems;
}
```

The **ds.verify** method's single parameter is an object that has functions as attributes. As the verification progresses, Wakanda Server calls back to the appropriate function(s) with information. In the example below, we are only interested in the call back for addProblem, which is called as the verification process discovers individual issues with the data or indices. When addProblem is called, it is provided a problem object containing information about the issue. In our example, we convert the object to a string value and write it to the project's log file after the heading "Verify: " so that when examining the log file later we can clearly see items concerning this routine. Notice that the log file used is simply "log.txt." In our project, we use this log file for a variety of tasks and therefore require the services of a mutex to make sure updates to the log are done in an atomic way.

There are several other potential call back methods supported by verify and we could have included these in problemHandler. These include openProgress(title, maxElements), setProgressTitle(title), progress(currentElementNumber, maxElements) and closeProgress(). For more information, see the verify( ) API.

openRemoteStore( )

Datastore **openRemoteStore**(Object *params*)

Parameter	Type	Description
params	Object	Parameters for the remote datastore access
Returns	Datastore	Reference to the remote datastore

Description

The **openRemoteStore( )** method opens the remote datastore defined by *params* in the current solution and returns a reference to it. The remote target datastore can be one of the following servers:

- another Wakanda Server
- a 4D Server (v14 and higher)
- an SQL database

In the *params* object, pass a set of properties that defines the remote datastore to which you want to connect. The required properties depend on the kind of datastore you want to open:

Property	Value type	Description	Wakanda Server	4D Server	SQL database
hostname	*String*	Full name or IP address of the remote server	X	X	X
ssl	*Boolean*	Enable SSL connection	X	X	X
user	*String*	User name	X	X	X
password	*String*	User password	X	X	X
SQL	*Boolean*	**true** to connect to an SQL server			X
port	*Number*	Port number			X
database	*String*	Database name			X

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. Since connection to the remote datastore is synchronous, the reference can be used in your code just after the **openRemoteStore( )** method call.

Note that the returned reference will be valid only for the current JavaScript context. Once the JavaScript context is closed, the datastore reference is removed from memory by the JavaScript garbage collector. If you want to create and maintain a reference during the whole working session, you may need to use the addRemoteStore( ) method instead.

Example

You want to connect to a remote Wakanda Server and get a collection:

```
var myWTarget =
    {   hostname: "www.mydb.com",
        ssl: true,
        user: "john",
        password: "lijo!d!_6"
    };
var targetDS = openRemoteStore(myWTarget);
var vTotal = targetDS.Invoice.all(); // access a class from the remote datas
```

Example

You want to connect to a remote SQL database:

```
var mySQLTarget =
    {   SQL: true
        hostname: "123.45.67.89",
        port: 19812,
        database: "Invoices",
        user: "jim",
        password: "pwd555"
    };
var targetSQL = openRemoteStore(mySQLTarget);
```

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 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).

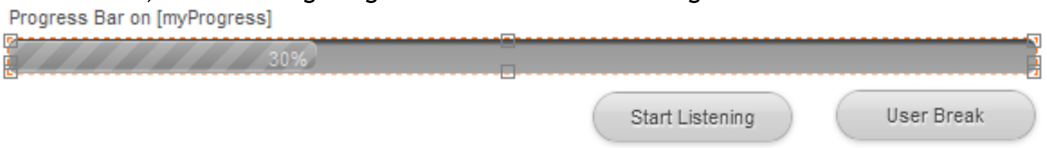
Example

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(10000000, "Processing element {curValue} ou
var s = "" ;
for (var i = 1; i  < 10000000; i++) // Processing loop
{
    if (prog.setValue(i))  // Increment progress indicator and test for
    {
        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:

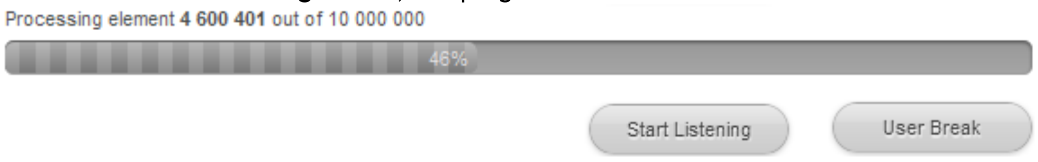
```
button1.click = function button1_click (event)
{
    $$("progressBar0").startListening(); // connect to the session c
};
```

__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:

```
button2.click = function button2_click (event)
 {
     $$("progressBar0").userBreak(); // send request to interrupt
 };
```

- You can then execute the code on the server and connect to it through your 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.

repairDataStore( )

void **repairDataStore**(File *model*, File *data*[, Object *options*], File *repairedData*)

Parameter	Type	Description
model	File	Datastore's model file to repair
data	File	Data file to repair
options	Object	Callback functions
repairedData	File	Destination for the repaired data file

Description

The **repairDataStore( )** method repairs the datastore's *data* file defined by *model* and *data*, and generates the *repairedData* data file.

- *model*: pass a reference to the datastore's model file (whose extension is ".waModel").
- *data*: specify a reference to *model*'s data file (whose extension is ".waData") to compact.
- *repairedData*: pass a reference to the repaired data file (whose extension must be ".waData") that will be created.

You can designate the current *model* file, but not the current *data* file. If you attempt to repair a data file that is currently open, an error will be generated. The *data* data file is opened in read only mode before it is repaired.

Make sure that the path for *repairedData* is valid and that you have enough space on your destination disk for the duplicated data file.

This method performs a standard repair and should only be used when a few entities or indexes are damaged. If you have any issue with the address tables, this method will not have an effect. Such information can be retrieved from the **verifyDataStore( )** method.

If the method encounters a level 2 error (standard error), it will repair the data file. If it encounters a level 1 error (fatal error), it stops executing. For more information about error levels, see the *options* parameter description.

options

In the *options* parameter, you can pass an object containing one or more callback functions. These functions will be called automatically by Wakanda's maintenance or backup engine (with the relevant parameter(s)) when the corresponding event occurs. You then need to write code to handle the information returned by these functions.

Note: *You can omit the options parameter when processing a data file, but if an error occurs, you will not know the reason why it occurred.*

The *options* parameter has the following callback functions:

- **openProgress: function(title, maxElements)**
  This function is called each time the engine starts processing a new group of elements or a new file, i.e., for each new **progress event**. For a maintenance operation, the event could be the processing of entities for a datastore class, an index table, etc. For a backup operation, it could be the backup of the journal file, the backup initialisation or the backup of the data file. This function is called with the following parameters:
  - *title* (string): label for the new progress event (e.g., "Verifying entities in datastore class #3").
  - *maxElements* (number): total number of elements to handle in the group (e.g., the number of entities in the datastore class).

  You can use this function, for example, to open a progress indicator on the client to display the progress or to write the progress of the operation to a log file on disk.

- **closeProgress: function()**
  This function is called when a progress event is closed. Progress events can be nested at multiple levels (e.g., one progress event can start checking all the indexes and another one opens for each index). A closeProgress event is generated for each openProgress event.

- **progress: function(curElement, maxElements)**
  This function is called during a progress event for each element to be verified, repaired, compacted (e.g., an entity or an index page) or backed up. This function is called with the following parameters:
  - *curElement* (number): position of the element currently being handled in the group.
  - *maxElements* (number): total number of elements to handle in the group (same value as in openProgress).

- **setProgressTitle: function(title)**
  This function is called when the title of a progress event has changed for some reason (e.g., when a missing piece of information is found while verifying the data). This function is called with the following parameter:
  - *title* (string): new label for the progress event.

  ***Note**: This function is rarely called.*

- **addProblem: function(problem)**
  This function is called when a problem has been detected in the data during a progress event. If an error occurs, only this function is called. It is therefore mandatory to call this function if you want to know whether the verification of a data file was successful or not. This function is called with the following parameter:
  - *problem* (object): object describing the problem. Information provided in this object depends on the type of the problem. For example, if an entity cannot be read, the entity number is returned in the *ErrorText* property.
    The *problem* object is returned with the following properties:
    - *problem.ErrorText* (string): complete description of the error
    - *problem.ErrorNumber* (number): error number
    - *problem.ErrorType* (number): internal code that returns the type of the error
    - *problem.ErrorLevel* (number): level of seriousness of the error. Available values are:
      - 1=fatal error (currently unused)
      - 2=regular error: serious errors including various kind of issues (wrong index type, invalid file header, checksum errors, data and model inconsistencies...). Some regular errors may require the datastore to be repaired.
      - 3=warning: minor errors, such as indexes that need to be rebuilt or externally stored blob field content that is not found.

Example

The following function shows an example of how a **repairDataStore( )** method call could be organized:

```
function demoRepair()
{
    function myOpenProgress(title, maxElements)
     {
         console.log(title+" on "+maxElements+" elements"); // log each new p
     }

    function myProgress(curElement, maxElements)
    {
        console.log("current element: "+curElement);  // log each element
            // events can be nested
    }

    function myCloseProgress()
    {
        ... // add code to handle the closing of the progress event
    }

    function myAddProblem(problem)
    {
        if (problem.ErrorLevel <= 2) // we only handle fatal or regular erro
        {
            this.storedProblems.push(problem);  // fill the custom array
            console.log(problem.ErrorText);  // log the error description
        }
    }

    var modelFile = File("c:/wakanda/mySolution/people.waModel");
    var modelData = File("c:/wakanda/mySolution/people.waData");
    var repairDest = File("c:/wakanda/mySolution/repairedData.waData");
    var options = {
        'openProgress': myOpenProgress,
        'closeProgress': myCloseProgress,
        'progress': myProgress,
        'addProblem': myAddProblem,

            // you can add any custom code here, it will be passed to the
            // addProblem function in the 'this' keyword
        'storedProblems':[] // we add an array to store any problems that ar
        }

    repairDataStore(modelFile, modelData, options, repairDest);

    if (options.storedProblems.length > 0)
    {
        ... // code to warn the user that some problems occurred
    }
    else
    {
        ... // everything is ok
    }
}
```

requestFileSystemSync( )

FileSystemSync **requestFileSystemSync**(Number *type*, Number *size*)

Parameter	Type	Description
type	Number	0 = Temporary, 1 = Persistent
size	Number	Expected storage space (in bytes)
Returns	FileSystemSync	New synchronous file system object

Description

The **requestFileSystemSync( )** method requests a *FileSystemSync* object in which application data can be stored. If the method is executed successfully, a new *FileSystemSync* object is returned.

Pass in *type* a value describing the type of storage space you want the file system to reserve. You can pass either a number value or a constant:

Constant	Value	Description
TEMPORARY	0	Used for storage with no guarantee of persistence. Data stored in a temporary storage may be deleted at the server's convenience, e.g. to deal with a shortage of disk space. In Wakanda, temporary storage is deleted after the Javascript context is closed.
PERSISTENT	1	Used for storage that should not be removed by the server without user permission. Data stored there will not deleted after the Javascript context is closed.

Pass in *size* a value indicating how much storage space, in bytes, the server expects to need.

Example

Create a temporary file system:

```
var myFileS = requestFileSystemSync(TEMPORARY, 1024*1024);
```

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*.

Pass in *id* the module identifier. Identifiers may refer to a file or a folder. For complete information about the *id* parameter and the CommonJS architecture, please refer to the CommonJS specification.

General case
Usually, a Wakanda CommonJS module is a .js (or .json) file stored in a **'Modules'** folder located at the root of the application (project) folder. In this case, you can just pass the file name (without the '.js' extension) in the *id* parameter. For example, if you want to reference a "tools.js" module file stored in the "Modules" folder fo your application, you can just write:

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

However, several options and mechanisms are supported regarding the *id* parameter.

Path of the file or folder
There are three types of module identifier pathes:

- *Relative to current module*: if the *id* parameter starts with "./" or "../", then the path to the module to load is relative to the current module.
  Suppose we are executing "test.js" in "/myFile/js".
  If we do **require('./myModule1')**, then myModule1 will be looked for here: "/myFile/js/myModule1"
  If we do **require("../myModule2")**, then myModule2 will be looked for here: "/myFile/myModule2" (parent folder).
- *Absolute path*: if the *id* parameter starts with "/" or "x:/" (x being a valid Windows drive letter), then it is an absolute path.
- *Relative path*: in all other cases, id is a relative path. By default, **require( )** will look for modules in the **Modules** folder of your project (see "General" case paragraph). However, you can use the **require.paths** array to define other locations.
  paths is a read/write attribute of the **require** method: it contains an array of paths which can be used to resolve identifiers *id*. The lowest numbered is the most prioritary. **paths** can be freely modified but accesses are always checked.
  By default in Wakanda, this attribute contains a single path to the modules subfolder of your a
  Suppose we have defined:

  ```
  require.paths[0] = "/mymodules";
  require.paths[1] = "/tmp/test_modules";
  ```

  If the following statement is executed:

  ```
  require("abc/mods/test");
  ```

  ... then Wakanda will first try to load "/mymodules/abc/mods/test". If no module is found, then Wakanda will try to load "/tmp/test_modules/abc/mods/test".

Using a folder
The *id* parameter can refer to a folder. In this case, Wakanda first tries to read the "main" attribute of "package.json", which must be a relative path. If it fails, Wakanda tries to read the "index.js" file.
Suppose we do **require("/tmp/test/module")** and that "/tmp/test/module" is a folder.

Wakanda first tries to load "/tmp/test/module/package.json" and to read its "main" attribute. If "main" contains "./myfile", then the module to load is "/tmp/test/module/myfile". Note that "myfile" does not have a suffix, it can be a json or js file. If there is no "package.json" file, Wakanda loads "/tmp/test /module/index.js".

The script file for a module is called as a function with two arguments: *exports* and *module*. The *exports* variable is an empty object, the module implementation will put inside the API it wishes to make available.

The module may wish to make a "typed" object (using a **new()** expression), this is possible by returning the exported object in **module.exports**. Whenever **module.exports** is defined, **require( )** will return it instead.

require( ) can be called recursively, in which case a partially filled *exports* object is returned. This does not work if **module.exports** is used. Note that **module.exports** is only taken in consideration when the module has been fully evaluated, so recursive **require( )** calls will always return the *exports* object.

The *module* variable is in a scope. It contains information about the module being loaded. It has the following attributes :

- id: The identifier used
- uri: A URI to the module.
- filename: Full path to module.
- loaded: True if the module has been fully evaluated (false in recursive call).
- exports: Object to export instead of exports.

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" chapter of the Wakanda Doc Center (for an overview of SSJS modules, refer to the **About SSJS Modules** section).

***Note: require( )** calls are not compatible with standard try/catch structures. **require( )** errors will not be intercepted by **catch** statements.*

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
```

resolveLocalFileSystemSyncURL( )

EntrySync **resolveLocalFileSystemSyncURL**(String *url*)

Parameter	Type	Description
url	String	URL to a local file in the filesystem
Returns	EntrySync	EntrySync object corresponding to the url

Description

The **resolveLocalFileSystemSyncURL( )** method allows the user to look up the filesystem for a file or directory referred to by a local *url*.

It returns an *EntrySync* object that you can handle with the methods and properties of the **EntrySync** theme.

restoreDataStore( )

Object **restoreDataStore**(File *manifest* , Folder *restoreFolder* [, Object *options*])

Parameter	Type	Description
manifest	File	Backup manifest file
restoreFolder	Folder	Destination folder for the restored data folder
options	Object	Block of callback functions
Returns	Object	Resulting object

Description

The **restoreDataStore( )** method allows you to restore a data folder using a specific backup *manifest* file.

The data folder is restored from the backup folder referenced in the manifest file and copied to the *restoreFolder* destination *Folder*. The application's journal (if any) is also transferred at the same level as the restored data folder.

***Note:** When restarting the datastore after a restoration, you need to make sure that the restored log file is placed at the right location (the one defined in the* **Journal Settings**)*.You may need to move the file to the right place manually before restarting the datastore.*

In *manifest*, pass a *File* object that references a valid backup manifest file. For more information about manifests, please refer to the **Backup Manifest** paragraph.

In *restoreFolder*, pass a *Folder* object whose *path* property leads to a folder where the data folder must be restored.
If for some reason the destination folder already contains a data folder, the latter will be renamed before the restoration takes place. This mechanism allows you to specify a damaged data folder location and replace it with a restored version, so that the application can be relaunched with no additional setting (except moving the journal file to the appropriate location, see below).

***Warning**: If you specify the application's folder as the restoration target, make sure that the application is not running.*

The **restoreDataStore( )** method returns an object that gives information about the restoration process:

```
{
    ok: boolean,
    dataFolder: Folder
}
```

- The *ok* property value is **true** if the restoration process was completed sucessfully. It is **false** if an error occurred and no exception was thrown.
- The *dataFolder* property value returns a reference to the previous data folder, in case the restoration was done in a folder that already contains a data folder (see *restoreFolder*

parameter description). This is usually the case when you restore a data folder in the current data folder location of the application.
If no folder was renamed in the operation, the *dataFolder* property value is **undefined**.

options

In the *options* parameter, you can pass an object containing one or more callback functions. These functions will be called automatically by Wakanda's maintenance or backup engine (with the relevant parameter(s)) when the corresponding event occurs. You then need to write code to handle the information returned by these functions.

***Note**: You can omit the options parameter when processing a data file, but if an error occurs, you will not know the reason why it occurred.*

The *options* parameter has the following callback functions:

- **openProgress: function(title, maxElements)**
  This function is called each time the engine starts processing a new group of elements or a new file, i.e., for each new **progress event**. For a maintenance operation, the event could be the processing of entities for a datastore class, an index table, etc. For a backup operation, it could be the backup of the journal file, the backup initialisation or the backup of the data file. This function is called with the following parameters:
  - *title* (string): label for the new progress event (e.g., "Verifying entities in datastore class #3").
  - *maxElements* (number): total number of elements to handle in the group (e.g., the number of entities in the datastore class).

  You can use this function, for example, to open a progress indicator on the client to display the progress or to write the progress of the operation to a log file on disk.

- **closeProgress: function()**
  This function is called when a progress event is closed. Progress events can be nested at multiple levels (e.g., one progress event can start checking all the indexes and another one opens for each index). A closeProgress event is generated for each openProgress event.

- **progress: function(curElement, maxElements)**
  This function is called during a progress event for each element to be verified, repaired, compacted (e.g., an entity or an index page) or backed up. This function is called with the following parameters:
  - *curElement* (number): position of the element currently being handled in the group.
  - *maxElements* (number): total number of elements to handle in the group (same value as in openProgress).

- **setProgressTitle: function(title)**
  This function is called when the title of a progress event has changed for some reason (e.g., when a missing piece of information is found while verifying the data). This function is called with the following parameter:
  - *title* (string): new label for the progress event.

  ***Note**: This function is rarely called.*

- **addProblem: function(problem)**
  This function is called when a problem has been detected in the data during a progress event. If an error occurs, only this function is called. It is therefore mandatory to call this function if you want to know whether the verification of a data file was successful or not. This function is called with the following parameter:
  - *problem* (object): object describing the problem. Information provided in this object depends on the type of the problem. For example, if an entity cannot be read, the entity number is returned in the *ErrorText* property.
    The *problem* object is returned with the following properties:

- *problem.ErrorText* (string): complete description of the error
- *problem.ErrorNumber* (number): error number
- *problem.ErrorType* (number): internal code that returns the type of the error
- *problem.ErrorLevel* (number): level of seriousness of the error. Available values are:
    - 1=fatal error (currently unused)
    - 2=regular error: serious errors including various kind of issues (wrong index type, invalid file header, checksum errors, data and model inconsistencies...). Some regular errors may require the datastore to be repaired.
    - 3=warning: minor errors, such as indexes that need to be rebuilt or externally stored blob field content that is not found.

Example

In this example, there is an existing data folder in the restoration folder.

Assuming the following folder structure:

```
|- /
|  -var
|      |
|      |-testManifest.json
|      |
|      |- testResultFolder
|      |         |
|      |         |- DataFolder
|      |
|      |- foobar
|      |         |
|      |         |- SomeFunkyFolder
```

If you execute the following code:

```
var result = restoreDataStore(File("/var/manifest.json"),Folder("/var/testRes
    // result.ok is true
    // result.dataFolder is Folder("/var/testResultFolder/DataFolder_REPLACEI
```

The folder structure becomes:

```
|- /
|  -var
|      |
|      |-testManifest.json
|      |
|      |- testResultFolder
|      |         |
|      |         |- DataFolder
|      |         |
|      |         |- DataFolder_REPLACED_2012-12-22_23-23-00
|      |
|      |- foobar
|      |         |
|      |         |- SomeFunkyFolder
```

Example

In this example, there is no existing data folder in the restoration folder.

With the same initial folder structure as below, if you execute the following code:

```
var result = restoreDataStore(File("/var/manifest.json"),Folder("/var/foobar/
    // result.ok is true
    // result.dataFolder is undefined
```

The resulting folder structure is:

```
|- /
|  -var
|     |
|     |-testManifest.json
|     |
|     |- testResultFolder
|     |        |
|     |        |- DataFolder
|     |        |
|     |
|     |- foobar
|     |        |
|     |        |- SomeFunkyFolder
|     |        |          |- DataFolder
```

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:** 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

This code will loop and log the loops:

```
function treatTimer() {
    console.log("treatTimer nbLoops=",nbLoops); //log the loop number
    nbLoops++;
    if (nbLoops < 5) {
        setTimeout(function() {treatTimer()},3000);
    }
    else {
        exitWait(); //stop the wait
    }
}

var a=1;
var nbLoops = 0;
setTimeout(function() {treatTimer()},3000); // call the function
a++;
console.log("a=",a,' "name"->',"GUY"+a);
a++;
a++;
wait();   // wait here until exitWait is called
```

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 the thread that spawned them 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 that the project folder is the default folder. 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 its function declarations, Wakanda considers it as 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 this existing shared worker.

Example

Download the example solution

This shared worker creates an entity every second for 5 seconds, and sends info 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;
            }
       }
     wait(); //waits for new messages in onmessage
}
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.port; //MessagePort
    thePort.onmessage = function(event)
    {
        var message = event.data;
        switch (message.type)
        {
            case 'connected':
                tmRef = message.ref;
                taskMgr.postMessage({type: 'report', ref: tmRef});
                break;

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

            case 'error':
                tmInfo.errorCode = message.errorCode;
                return tmInfo;
                close();
                break;
        }
    }
    wait();//waits until a call to close() in this thread
    //allows to handle incoming messages on the onmessage
    //at this point, this thread is about to end but the shared
    //worker continues on
}
```

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* [, String | Folder *executionPath*])

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

Description

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

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

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

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

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

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

Using built-in commands

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

For example:

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

Note about returned data

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

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

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

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

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

Example with the 'dir' command on Windows:

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

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

Example

The following 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

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

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

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

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

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

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

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

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

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

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.

Note that **TextStream( )** always uses a CRLF combination for line breaks.

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 ir
                stream.write(myMessage+"\n"); // append the message to the er
                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();
```

verifyDataStore( )

void **verifyDataStore**(File *model*, File *data*, Object *options*)

Parameter	Type	Description
model	File	Datastore's model file to verify
data	File	Data file to verify
options	Object	Block of callback functions

Description

The **verifyDataStore( )** method verifies the internal structure of the objects contained in the datastore designated by *model* and *data*.

- *model*: pass a reference to the datastore's model file (whose extension is ".waModel").
- *data*: specify a reference to *model*'s data file (whose extension is ".waData") to check.

You can designate the current *model* file, but not the current *data* file. If you attempt to verify a data file that is currently open, an error will be generated. The data file designated is opened in read only mode. Before calling this method, make sure that no application has access to this file in write mode because the results of this data file verification may be distorted.

options

In the *options* parameter, you can pass an object containing one or more callback functions. These functions will be called automatically by Wakanda's maintenance or backup engine (with the relevant parameter(s)) when the corresponding event occurs. You then need to write code to handle the information returned by these functions.

***Note**: You can omit the options parameter when processing a data file, but if an error occurs, you will not know the reason why it occurred.*

The *options* parameter has the following callback functions:

- **openProgress: function(title, maxElements)**
  This function is called each time the engine starts processing a new group of elements or a new file, i.e., for each new **progress event**. For a maintenance operation, the event could be the processing of entities for a datastore class, an index table, etc. For a backup operation, it could be the backup of the journal file, the backup initialisation or the backup of the data file. This function is called with the following parameters:
    - *title* (string): label for the new progress event (e.g., "Verifying entities in datastore class #3").
    - *maxElements* (number): total number of elements to handle in the group (e.g., the number of entities in the datastore class).

  You can use this function, for example, to open a progress indicator on the client to display the progress or to write the progress of the operation to a log file on disk.

- **closeProgress: function()**
  This function is called when a progress event is closed. Progress events can be nested at multiple levels (e.g., one progress event can start checking all the indexes and another one opens for each index). A closeProgress event is generated for each openProgress event.

- **progress: function(curElement, maxElements)**
  This function is called during a progress event for each element to be verified, repaired, compacted (e.g., an entity or an index page) or backed up. This function is called with the following parameters:
  - *curElement* (number): position of the element currently being handled in the group.
  - *maxElements* (number): total number of elements to handle in the group (same value as in openProgress).

- **setProgressTitle: function(title)**
  This function is called when the title of a progress event has changed for some reason (e.g., when a missing piece of information is found while verifying the data). This function is called with the following parameter:
  - *title* (string): new label for the progress event.

  ***Note**: This function is rarely called.*

- **addProblem: function(problem)**
  This function is called when a problem has been detected in the data during a progress event. If an error occurs, only this function is called. It is therefore mandatory to call this function if you want to know whether the verification of a data file was successful or not. This function is called with the following parameter:
  - *problem* (object): object describing the problem. Information provided in this object depends on the type of the problem. For example, if an entity cannot be read, the entity number is returned in the *ErrorText* property.
    The *problem* object is returned with the following properties:
    - *problem.ErrorText* (string): complete description of the error
    - *problem.ErrorNumber* (number): error number
    - *problem.ErrorType* (number): internal code that returns the type of the error
    - *problem.ErrorLevel* (number): level of seriousness of the error. Available values are:
      - 1=fatal error (currently unused)
      - 2=regular error: serious errors including various kind of issues (wrong index type, invalid file header, checksum errors, data and model inconsistencies...). Some regular errors may require the datastore to be repaired.
      - 3=warning: minor errors, such as indexes that need to be rebuilt or externally stored blob field content that is not found.

Example

The following function shows the various ways to manage information provided by the **verifyDataStore( )** method:

```
function demoVerify()
{
    function myOpenProgress(title, maxElements)
     {
         console.log(title+" on "+maxElements+" elements"); // log each new p
     }

    function myProgress(curElement, maxElements)
    {
        console.log("current element: "+curElement);  // log each element
            // events can be nested
    }

    function myCloseProgress()
```

```
        {
            ... // add code to handle the closing of the progress event
        }

        function myAddProblem(problem)
        {
            if (problem.ErrorLevel <= 2) // we only handle fatal or regular erro
            {
                this.storedProblems.push(problem);  // fill the custom array
                console.log(problem.ErrorText);  // log the error description
            }
        }

        var modelFile = File("c:/wakanda/mySolution/people.waModel");
        var modelData = File("c:/wakanda/mySolution/people.waData");

        var options = {
            'openProgress': myOpenProgress,
            'closeProgress': myCloseProgress,
            'progress': myProgress,
            'addProblem': myAddProblem,

                // you can add any custom code here, it will be passed to the
                // addProblem function in the 'this' keyword
            'storedProblems':[] // we add an array to store any problems that ar
            }

        verifyDataStore(modelFile, modelData, options);

        if (options.storedProblems.length > 0)
        {
            ... // code to warn the user that some problems occurred
        }
        else
        {
            ... // everything is ok
        }
    }
```

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 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 then 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 the *Worker* type. 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 the thread that spawned them 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 that the project folder is the default folder. 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 its function declarations, Wakanda considers it as initialization code for the worker and executes it when the worker is created.*

Example

The following is a simple example of a 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 fo
// This will return a proxy object (worker) so that it will be possible to c
var worker = new Worker('child.js');

// Define the message callback that will be triggered each time the child sen
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 t
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. 
            // 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
        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, by default the method automatically uses the system proxy settings. However, if you want to override your system settings, you can pass an object containing two attributes in the *proxy* parameter:

- 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:

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

If you do not want to use your proxy for the request, pass an empty object in *proxy*.
***Note: XMLHttpRequest( )** supports HTTPS connections but does not validate certificates.*

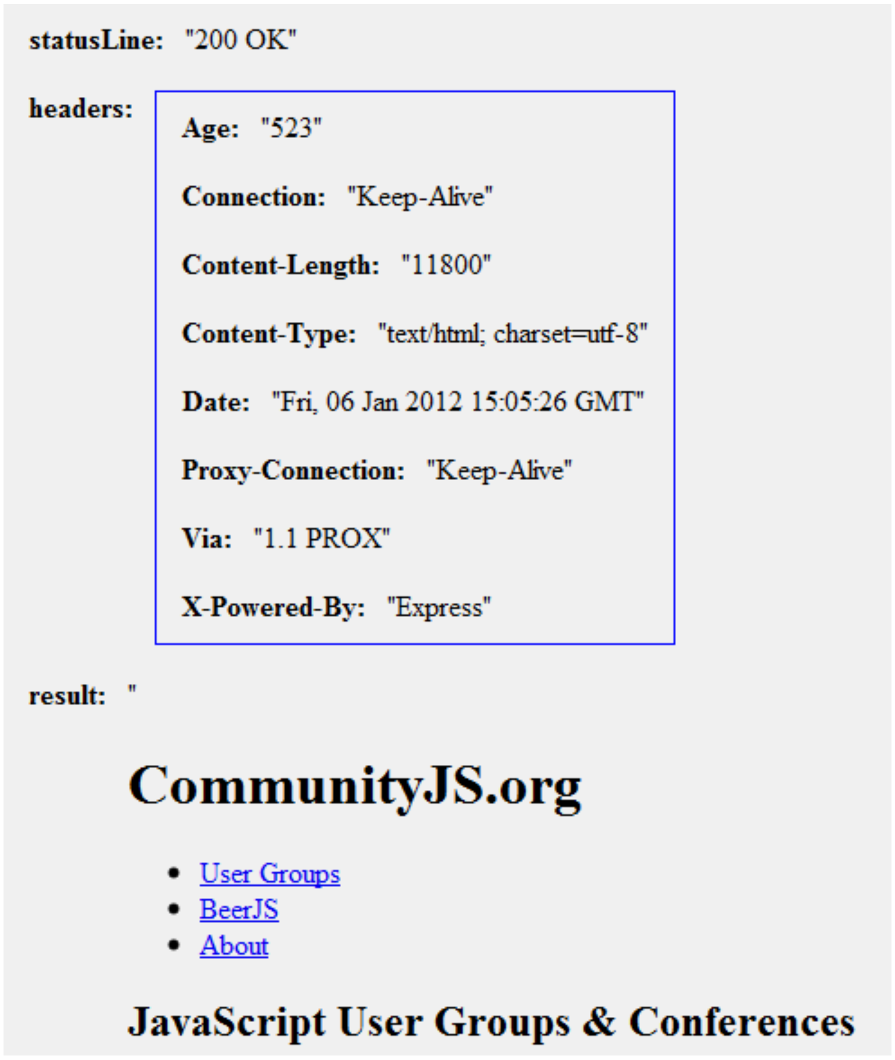
Example

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;

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

 xhr = new XMLHttpRequest(); // instanciate the xhr object
    // you could pass a proxy parameter if you do not want to use your defau

 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 re
     var result = this.responseText;  //get the contents of the response
     var headersArray = headers.split('\n'); // split and format the headers
     headersArray.forEach(function(header, index, headersArray) {
         var name, indexSeparator, value;

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

        indexSeparator = header.indexOf(':');
        name = header.substr(0,indexSeparator);
        if (name === "") {
            return;
        }
        value = header.substr(indexSeparator + 1).trim(); // clean up the he
        headersObj[name] = value; // fills an object with the headers
     });
     if (headersObj['Content-Type'] && headersObj['Content-Type'].indexOf('j
             // 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 edito
 ({
     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:



statusLine: "200 OK"

headers:

Age: "523"

Connection: "Keep-Alive"

Content-Length: "11800"

Content-Type: "text/html; charset=utf-8"

Date: "Fri, 06 Jan 2012 15:05:26 GMT"

Proxy-Connection: "Keep-Alive"

Via: "1.1 PROX"

X-Powered-By: "Express"

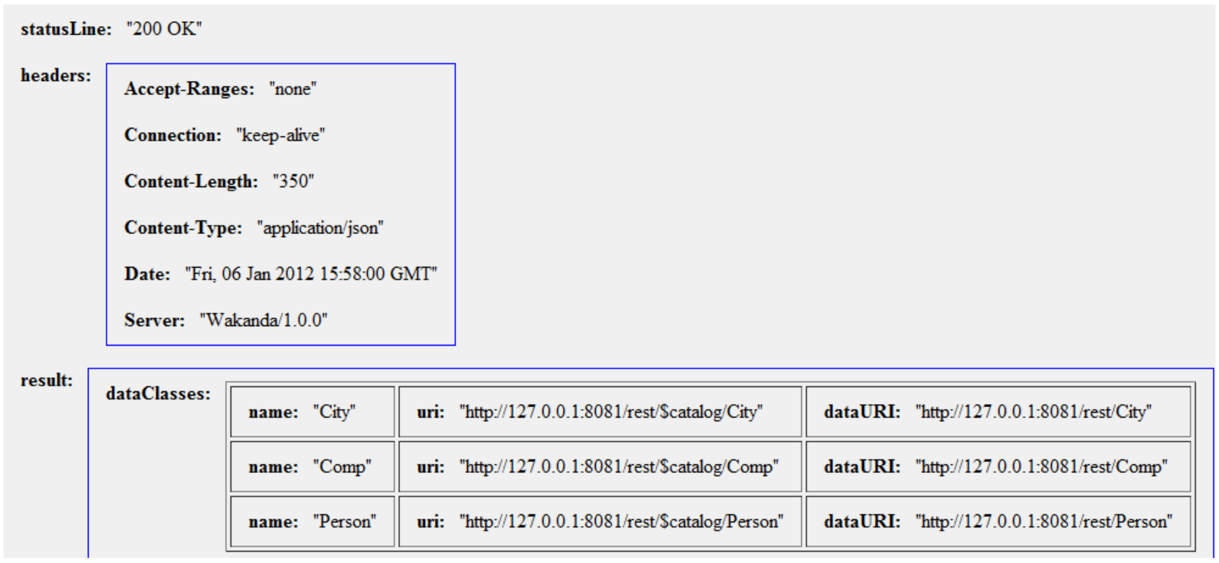
result: "

CommunityJS.org

- User Groups
- BeerJS
- About

JavaScript User Groups & Conferences

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



statusLine: "200 OK"

headers:

Accept-Ranges: "none"

Connection: "keep-alive"

Content-Length: "350"

Content-Type: "application/json"

Date: "Fri, 06 Jan 2012 15:58:00 GMT"

Server: "Wakanda/1.0.0"

result:

dataClasses:

name: "City"	**uri:** "http://127.0.0.1:8081/rest/$catalog/City"	**dataURI:** "http://127.0.0.1:8081/rest/City"
name: "Comp"	**uri:** "http://127.0.0.1:8081/rest/$catalog/Comp"	**dataURI:** "http://127.0.0.1:8081/rest/Comp"
name: "Person"	**uri:** "http://127.0.0.1:8081/rest/$catalog/Person"	**dataURI:** "http://127.0.0.1:8081/rest/Person"

XmlToJSON( )

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

Parameter	Type	Description
xmlText	String	XML formatted string
jsonFormat	String	JSON format to use
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.

In the *xmlText* parameter, pass a valid XML string to convert. By default, if you omit the *jsonFormat* parameter, **XmlToJSON( )** will use a default format that produces a comprehensive representation of the initial XML string.

To obtain a more compact result, you can pass a string representing the JSON format to use in the *jsonFormat* parameter. Currently, only the "json-bag" format option is supported.

If you pass a value in *jsonFormat*, you must pass the name of the root element of the XML string (or an empty string) in the *rootElement* parameter. If you omit this parameter, a "bag" element is expected as the root element. If the root element is not found, an empty json object {} is returned.

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
```

Example

The following example shows the difference between the default formatting (more complete) and the 'json-bag' formatting (more compact):

```
var myXML =  "<people><employee><name>Smith</name><job>Developer</job></emplo
var myFullJSON = XmlToJSON(myXML); // default format
var myCompactJSON = XmlToJSON(myXML,"json-bag","people"); // json-bag format

// myFullJSON
// "{"document":["childNodes":[{"nodeType":1,"nodeName":"people","childNodes'
// [{"nodeType":1,"nodeName":"name","childNodes":[{"nodeType":3,"nodeValue":'
// [{"nodeType":3,"nodeValue":"Developer"}]}]},{"nodeType":1,"nodeName":"empl
// [{"nodeType":3,"nodeValue":"Jones"}]},{"nodeType":1,"nodeName":"job","chil
// {"nodeType":1,"nodeName":"customer","attributes":{"name":"Wesson"}}]}]}"

// myCompactJSON
// "{"employee":[{"name":[{"__CDATA":"Smith"}],"job":[{"__CDATA":"Developer"}
// [{"__CDATA":"Secretary"}]}],"customer":[{"name":"Wesson"}]}"
```

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. An application is a Wakanda running project. For more information about the Application objects, refer to the Application class description.

name

Description

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

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**(String *projectName*)

Parameter	**Type**	**Description**
projectName	String	Name of the project
Returns	Object	Application object

Description

The **getApplicationByName( )** method returns a reference to the application object whose name you passed in the *projectName* parameter - an application is a running Wakanda project. The application/project 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

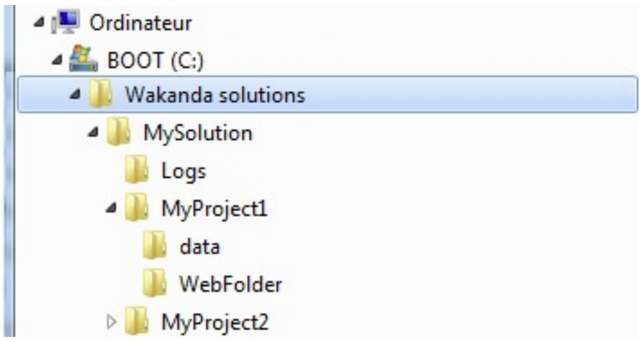
The **getFolder( )** method returns the folder containing the solution file (named '*SolutionName*.waSolution').

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 mo
```

getItemsWithRole( )

File **getItemsWithRole**(String *role*)

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

Description

The **getItemsWithRole( )** method returns the solution-level file associated with the *role* you passed as a parameter.

The method returns a *File* object.

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

role	Type	Description
"settings"	File	XML file defining the settings for your solution(*solutionName*.waSettings). See **Solution Settings File**.
"directory"	File	Directory file for the solution (*solutionName*.waDirectory). See **Users and Groups**.
"permissions"	File	Permission file for the solution (*solutionName*.waPerm). See **Assigning Group Permissions**.

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 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}...
```

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.