Files and Folders

The "Files and Folders" theme provides an interface with the server's OS File system.

Files and Folders objects allow you to handle files and folders as JavaScript objects using specific classes. Wakanda also provides support for reading and writing text streams and binary streams.

Wakanda Server also implements the **FileSystem API** W3C specification, originally designed for a client application (user agent). Using this API, you can navigate and change file system hierarchies.

In this documentation, this API is listed in themes starting with "W3C". For more information on the W3C specification, please refer to the File API: Directories and System working draft.

This API is compatible with the Wakanda server Files and Folders native API, that is, you can get a valid **File( )** or **Folder( )** object from browsing a FileSystem or a **FileSystemSync** object.

Note: *Wakanda supports* asynchronous *FileSystem calls, but this documentation only describes the synchronous interface calls. Please refer to the W3C draft for more details on the asynchronous calls.*

Using FileSystem Objects

Wakanda Server implements **FileSystem** objects (see the **Files and Folders** introduction). In Wakanda, you can use these objects to define a set of folders externally where SSJS or the server will have access by means of a relative path in a sandboxed environment.

Basically, we recommend that you use FileSystem objects whenever you need to reference files or folders in your Wakanda applications. This feature is useful for two main reasons:

- independence: You can move your solution from one place to another, regardless of the OS, without having to worry about file and folder paths,
- security: No JavaScript code can access elements located above the Filesystem roots on the disk (sandboxing).

As described in the **requestFileSystemSync( )** constructor documentation, Wakanda supports two standard *FileSystem* object types: PERSISTENT and TEMPORARY.

Wakanda also supports custom FileSystem objects based on definitions that you can add in a *fileSystem.json* file.

FileSystem Definition File

In Wakanda, you can define *FileSystem* root locations in a specific file named "**fileSystems.json**". Within this file, you must use the following syntax:

```
[
    {
            "name":"TEST",
            "path":"c:\\temp\\"
    },
    {
            "name":"SRV",
            "path":"\\\\srv-company\\TEMPO\\MYTEMP\\"
    }
]
```

- The "name" property must not be empty and must not contain any '/' characters.
- The "path" property must be an absolute path in POSIX notation (MS-DOS notation is also accepted on Windows). It can also be a relative POSIX path if a 'parent' JSON property is specified (see the "Relative FileSystems" paragraph below).

***Implementation Note**: In current Wakanda versions, the "writable" FileSystem property is not supported. In future releases, this property will be handled and if 'false', it will trigger an exception in the case of a write attempt by a JavaScript API.*

When Wakanda Server starts, FileSystem object definitions are loaded and combined from **fileSystems.json** file(s) located:

- in the project folder,
- and/or in the solution folder,
- and/or in the Wakanda resources folder.

In SSJS code, *FileSystem* objects can be instantiated using:

- the **requestFileSystemSync( )** function
- the **FileSystemSync( )** constructor

The following example code uses the *fileSystem.json* file defined above to access a text file located in \\srv-company\TEMPO\MYTEMP\test.txt:

```
var fs = FileSystemSync( "SRV");
var rootFolder = fs.root;
var theFile = File( rootFolder, "test.txt");
var theFileContents = theFile.toString();
```

Since **File( )** accepts a *FileSystem* object directly as first parameter, you can write the code above as follows:

```
var theFileContents = File( FileSystemSync("SRV"), "test.txt").toString();
```

Relative FileSystems

In the **fileSystems.json** definition file, you can define a *FileSystem* relative to another one using the JSON 'path' and 'parent' attributes:

- The 'path' attribute must be a POSIX relative folder path.
- The 'parent' attribute must be the name of an already-defined *FileSystem*.

Example of how to specify another name for the data folder in the current project folder (PROJECT *FileSystem*):

```
[
    {
            "name":"DATA",
            "parent":"PROJECT",
            "path":"myOtherData"
    }
]
```

Modifying the DataFolder Default Location

When opening a project, if a "DATA" *FileSystem* has been defined in the **fileSystems.json** file located in the **project** folder, Wakanda will use this *FileSystem* folder location for the data folder, regardless of what might be defined in the project definition file.

This feature allows the administrator to change the data location without modifying the project file itself.
It also allows you to specify a data location outside of the project folder.

Example using an absolute path:

```
[
    {
            "name":"DATA",
            "path":"c:\\myData\\"
    }
]
```

Example using a path relative to the project folder (PROJECT FileSystem):

```
[
    {
            "name":"DATA",
            "parent":"PROJECT",
            "path":"../../myDataFolder"
    }
]
```

Built-in FileSystems

Wakanda automatically defines several *FileSystem* objects when starting up:

- THEMES_CUSTOM maps to /.USER_HOME_DIRECTORY/Wakanda/Themes (read-only)
- WIDGETS_CUSTOM maps to /.USER_HOME_DIRECTORY/Wakanda/Widget (read-only)
- WALIB maps to the walib folder inside Wakanda Server in read-only

- THEMES_CUSTOM and WIDGETS_CUSTOM are defined in the *fileSystems.json* file located in the Wakanda Server resources folder.

When opening a solution:

- SOLUTION maps to the solution folder in read-only

And when opening each project:

- PROJECT maps to the project folder in read-only
- DATA maps to the data folder in read-write
- WEBFOLDER maps to the web folder in read-only

Syntax for FileSystems

According to W3C specifications, a syntax for using FileSystem objects in your code should go through several steps. For example, the following code uses the *fileSystem.json* file defined above to access a text file located in \\srv-company\TEMPO\MYTEMP\test.txt:

```
var fs = FileSystemSync( "SRV");
var rootFolder = fs.root;
var theFile = File( rootFolder, "test.txt");
var theFileContents = theFile.toString();
```

Wakanda fully supports this academic syntax.
However, for convenience, Wakanda allows you to combine the FileSystem and the path relative to this FileSystem in a single string. We recommend using this syntax in your Wakanda applications since it simplifies FileSystem manipulation:

```
/{FILESYSTEM_NAME}/{RELATIVE_PATH}
```

For example, you can write:

```
var myFile = File( "/PROJECT/untitled.txt"); //create the file reference in the project folder
```

This syntax is recognized by the **File( )** and **Folder( )** constructors when passing a single parameter of the string type, and wherever Wakanda expects a path.

Example

You want to access pictures located in a "Personnel_Photos" folder stored in your project folder.
First you need to define the "Personal_Photos" folder as a FileSystem in the **fileSystems.json** file of your project folder. To do this, you write:

```
[
    {
            "name":"PHOTOS",
            "parent":"PROJECT",
            "path":"Personal_Photos"
    }
]
```

Then you can access elements inside this FileSystem using "/PHOTOS":

```
var p = new ds.Friend();
p.name = "Sarah";
p.photo = "/PHOTOS/sarah.jpg";
p.save();
```

This code will work even if the project folder is moved.

BinaryStream

Properties and methods in the BinaryStream class allow you to handle and parse binary streams (i.e., streams of bytes).

To create a *BinaryStream* object, you need to execute the **BinaryStream( )** constructor method (from the Application class).

A *BinaryStream* object provides sequential access to binary data streams, mapped on binary files stored on disk.

Note that Wakanda also proposes support for *TextStream* objects for which you can define a *charSet* parameter.

BinaryStream objects can be built upon *Socket* or *SocketSync* objects. In this context:

- All *BinaryStream* methods throw an error in case of timeout, except **getBuffer( )** and **getBlob( )**.
- In case of timeout, **getBuffer( )** and **getBlob( )** methods report the amount of bytes actually read (can be zero).
- **getSize( )** returns the amount of bytes that have been read or written.
- **setPos( )** and **getPos( )** are not available.

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" for writing data, "Read" for reading data (default)
timeOut	Number	Timeout in milliseconds (used with 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 **Socket Instances** or **SocketSync Instances** sections.
  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())
```

changeByteOrder( )

void **changeByteOrder**( )

Description

The **changeByteOrder( )** method indicates that the next reading of structured values in the *BinaryStream* object requires a byte swap.

This method is useful when you want to read structured data (like long or real values -- actually any data type other that bytes) and the file used for the streaming was created on a platform where the byte ordering is different. The byte swap is necessary for example between PowerPC-based and Intel-based computers.

close( )

void **close**( )

Description

The **close( )** method closes the file referenced in the *BinaryStream* object.
The referenced file is opened when you execute the **BinaryStream( )** method and remains open until you call **close( )**.

flush( )

void **flush**( )

Description

The **flush( )** method saves the buffer contents to the disk file referenced in the *BinaryStream* object.
When you do several method calls to write data into a *BinaryStream* object, for optimization reasons the data is stored in a buffer that is saved to disk when the stream is closed. This method allows you to store the buffer during the process without having to close the stream.

getBlob( )

Blob **getBlob**(Number *sizeToRead*)

Parameter	Type	Description
sizeToRead	Number	Number of bytes to read
Returns	Blob	New BLOB object

Description

The **getBlob( )** method creates a new *BLOB* object containing the next *sizeToRead* data in the *BinaryStream* object, and moves the cursor to the next position in the stream.
Pass in *sizeToRead* the size of data to retrieve from the BinaryStream object. If you pass 0 (zero) in this parameter, the resulting BLOB size will be 0. If the data read from the BinaryStream object is < *sizeToRead* (end of file has been reached), the resulting BLOB will be resized accordingly. You have to call the **size** BLOB property to get the actual size of data.
If an error occurs, an exception is thrown and the method returns **Null**.

If the stream was built upon a *Socket* or *SocketSync* object, **getBlob( )** will not throw an error if the timeout is reached, and the data actually read is returned. It could be less than the *sizeToRead* value, or even zero. This allows you to read data without knowing its size in advance.

getBuffer( )

Buffer **getBuffer**(Number *sizeToRead*)

Parameter	Type	Description
sizeToRead	Number	Number of bytes to read
Returns	Buffer	New Buffer object

Description

The **getBuffer( )** method returns a new *Buffer* object containing the next *sizeToRead* data in the *BinaryStream* object, and moves the cursor to the next position in the stream.
Pass in *sizeToRead* the size of data to retrieve from the BinaryStream object. If you pass 0 (zero) in this parameter, the resulting Buffer size will be 0. If the data read from the BinaryStream object is < *sizeToRead* (end of file has been reached), the resulting Buffer will be resized accordingly. You have to call the **length** Buffer property to get the actual size of data.
If an error occurs, an exception is thrown and the method returns **Null**.

If the stream was built upon a *Socket* or *SocketSync* object, **getBuffer( )** will not throw an error if the timeout is reached, and the data actually read is returned. It could be less than the *sizeToRead* value, or even zero. This allows you to read data without knowing its size in advance.

getByte( )

Number **getByte**( )

Returns	Number	Byte at the cursor location

Description

The **getByte( )** method returns a number representing the next byte from the *BinaryStream* object and moves the cursor to the next position in the stream.

getLong( )

Number **getLong**( )

Returns	Number	Long value in the stream

Description

The **getLong( )** method returns the next long number (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.
Long numbers are coded on four (4) bytes. Note that the size and byte ordering of long numbers may vary from one OS to another.

getLong64( )

Number **getLong64**( )

Returns	Number	Next Long64 value in the stream

Description

The **getLong64( )** method returns the next long64 number (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

Long64 numbers are coded on eight (8) bytes. Note that the size and byte ordering of long numbers may vary from one OS to another.

getPos( )

Number **getPos**( )

Returns	Number	Position of the cursor in the stream

Description

The **getPos( )** method returns the current position of the cursor in the *BinaryStream* object.

Note: *This method cannot be used if the* ***BinaryStream*** *object has been built upon a* ***Socket*** *or* ***SocketSync*** *object.*

getReal( )

Number **getReal**( )

Returns	Number	Next real value in stream

Description

The **getReal( )** method returns the next real (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

getSize( )

Number **getSize**( )

Returns	Number	Current size of the stream (in bytes)

Description

The **getSize( )** method returns the size of the stream.
When you are reading the stream, this method returns the size of the file referenced in the *BinaryStream* object.
When you are writing the stream, this method returns the current size of the stream in memory.

When the stream is built upon a socket object, **getSize( )** allows you to know how much data was read or written.

getString( )

String **getString**

Returns	String	Next string value in the stream

Description

The **getString( )** method returns the next string (if present) from the *BinaryStream* object and moves the cursor to the next position in the stream.

getWord( )

Number **getWord**( )

Returns	Number	Next binary integer

Description

The **getWord( )** method returns the next word, i.e., a binary integer (if present) from the *BinaryStream* object, and moves the cursor to the next position in the stream.
Word numbers are coded on two (2) bytes. Note that the size and byte ordering of integers may vary from one OS to another.

isByteSwapping( )

Boolean **isByteSwapping**( )

Returns	Boolean	True if the stream needs byte-swapping

Description

The **isByteSwapping( )** method returns *True* if the current data reading in the *BinaryStream* object is in byte swap mode.
The byte swap mode is activated using the **changeByteOrder( )** method.

putBlob( )

void **putBlob**(Blob *blob* [, Number *offset*[, Number *size*]])

Parameter	Type	Description
blob	Blob	BLOB from which to read data
offset	Number	Byte offset where to start reading
size	Number	Number of bytes to read

Description

The **putBlob( )** method writes the *BLOB* you passed as the *blob* parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.
Optionally, you can pass in ***offset*** the byte position from which to start reading in the *blob*, and in ***size*** the number of bytes to read. By default, if you omit

these parameters, the whole ***blob*** contents is written (from 0 until the end).

If an error occurs, an exception is thrown.

putBuffer( )

void **putBuffer**(Buffer *buffer* [, Number *offset*[, Number *size*]])

Parameter	Type	Description
buffer	Buffer	Buffer from which to read data
offset	Number	Byte offset where to start reading
size	Number	Number of bytes to read

Description

The **putBuffer( )** method writes the ***Buffer*** you passed as the ***buffer*** parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

Optionally, you can pass in ***offset*** the byte position from which to start reading in the ***buffer***, and in ***size*** the number of bytes to read. By default, if you omit these parameters, the whole ***buffer*** contents is written (from 0 until the end).

If an error occurs, an exception is thrown.

putByte( )

void **putByte**(Number *byteValue*)

Parameter	Type	Description
byteValue	Number	Byte value to write into the stream

Description

The **putByte( )** method writes the byte value you passed as the parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

putLong( )

void **putLong**(Number *longValue*)

Parameter	Type	Description
longValue	Number	Long value to write into the stream

Description

The **putLong( )** method writes the long value you passed as the parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

putLong64( )

void **putLong64**(Number *long64Value*)

Parameter	Type	Description
long64Value	Number	Long64 value to write into the stream

Description

The **putLong64( )** method writes the long64 value you passed as the parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

putReal( )

void **putReal**(Number *realValue*)

Parameter	Type	Description
realValue	Number	Real value to write into the stream

Description

The **putReal( )** method writes the real value you passed as the parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

putString( )

void **putString**(*url*)

Parameter	Type	Description
		String value to write into the stream

Description

The **putString( )** method writes the string value you passed as the parameter in the ***BinaryStream*** object at the current cursor location. The cursor is then moved to the next position in the stream.

putWord( )

void **putWord**(Number *wordValue*)

Parameter	Type	Description
wordValue	Number	Word (integer) value to write into the stream

Description

The **putWord( )** method writes the byte word (i.e., an integer value) you passed as the parameter in the *BinaryStream* object at the current cursor location. The cursor is then moved to the next position in the stream.

setPos( )

void **setPos**(Number *offset*)

Parameter	Type	Description
offset	Number	New cursor position in the stream

Description

The **setPos( )** method moves the stream cursor to the position you passed in *offset* in the *BinaryStream* object.

Note: *This method cannot be used if the* *BinaryStream* *object has been built upon a* *Socket* *or* *SocketSync* *object.*

DirectoryEntrySync

A *DirectoryEntrySync* object represents a directory on a filesystem. This class inherits from the *EntrySync* class.

createReader( )

DirectoryReaderSync **createReader**( )

Returns	DirectoryReaderSync	New DirectoryReaderSync to read EntrySyncs

Description

The **createReader( )** method creates a new *DirectoryReaderSync* object to read entries from the *DirectorySync* to which it is applied.
For more information, please refer to the **DirectoryReaderSync** class description.

folder( )

Folder **folder**( )

Returns	Folder	Folder referenced by the directory entry

Description

***Note:** This method is not part of the W3C specification.*
The **folder( )** method returns a *Folder* object that represents the current state of the folder referenced by the *DirectoryEntrySync*.
The returned *Folder* object is a regular Wakanda object, that you can handle using the properties and methods of the **Folder**.

getDirectory( )

DirectoryEntrySync **getDirectory**(String *path* [, Object *options*])

Parameter	Type	Description
path	String	Absolute or relative path from the DirectoryEntrySync to the directory to be looked up or created
options	Object	Options for the directory creation
Returns	DirectoryEntrySync	Entry on the directory to be looked up or created

Description

The **getDirectory( )** method creates or looks up a directory and returns a new entry to it.
Pass in *path* an absolute path or a relative path from the current *DirectoryEntrySync* to the directory to be looked up or created.
Pass in the *options* parameter an object that can contain one or two of the following members:

options member	Type	Description
create	Boolean	If **true**, indicate that the user wants to create a file or directory if it was not previously there
exclusive	Boolean	By itself, *exclusive* has no effect. Passed along with *create*, it causes **getDirectory( )** and **getFile( )** to fail if the target path already exists

getDirectory( ) will fail if:

- *create* and *exclusive* are both **true** and the *path* already exists
- *create* is **not true** and the *path* does not already exist
- *create* is **not true** and the *path* exists, but is a file

If *create* is **true** and the *path* does not already exist, **getDirectory( )** creates and returns a corresponding *DirectoryEntrySync*.
Otherwise, if no other error occurs, **getDirectory( )** returns a *DirectoryEntrySync* corresponding to *path*.
***Note:** If you try to create a directory whose immediate parent does not yet exist, an error is generated.*

getFile( )

FileEntrySync **getFile**(String *path* [, Object *options*])

Parameter	Type	Description
path	String	Absolute or relative path from the DirectoryEntrySync to the file to be looked up or created
options	Object	Options for the file creation
Returns	FileEntrySync	Entry on the file to be looked up or created

Description

The **getFile( )** method creates or looks up a file and returns a new entry to it.
Pass in *path* an absolute path or a relative path from the current *DirectoryEntrySync* to the file to be looked up or created.
Pass in the *options* parameter an object that can contain one or two of the following members:

options member	Type	Description
create	Boolean	If **true**, indicate that the user wants to create a file or directory if it was not previously there
exclusive	Boolean	By itself, *exclusive* has no effect. Passed along with *create*, it causes **getDirectory( )** and **getFile( )** to fail if the target path already exists

getFile( ) will fail if:

- *create* and *exclusive* are both **true** and the *path* already exists
- *create* is **not true** and the *path* does not already exist
- *create* is **not true** and the *path* exists, but is a directory

If *create* is **true** and the *path* does not already exist, **getFile( )** creates it as a zero-length file and returns a corresponding *FileEntrySync*.
Otherwise, if no other error occurs, **getFile( )** returns a *FileEntrySync* corresponding to *path*.
Note: *If you try to create a file whose immediate parent does not yet exist, an error is generated.*

removeRecursively( )

void **removeRecursively**( )

Description

The **removeRecursively( )** method deletes the directory and all of its contents, if any.
In the event of an error (e.g. trying to delete a directory that contains a file that cannot be removed), some of the contents of the directory may be deleted.
If you try to delete the root directory of a filesystem, an error is generated.

DirectoryReaderSync

Objects of this class allow you to list files and directories in a directory.

readEntries( )

Array **readEntries**( )

Returns	Array	Array of EntrySync objects

Description

The **readEntries( )** method returns the next block of entries in the directory. It allows you to loop into a *DirectoryEntry* and build a list of its contents.
If there are no additions to or deletions from a directory between the first and last call to **readEntries( )**, and if no errors occur, then:

- A sequence of calls to **readEntries( )** returns each entry in the directory exactly once.
- Once all entries have been returned, the next call to **readEntries( )** returns an empty array.
- The array returned by **readEntries( )** is not empty while not all entries have been returned.
- The entries produced by **readEntries( )** does not include the directory itself ["."] or its parent [".."].

EntrySync

An object of the EntrySync type is an abstract interface representing entries in a file system, each of which may be a *FileEntrySync* or *DirectoryEntrySync*.

Both **DirectoryEntrySync** and **FileEntrySync** objects inherit from the **EntrySync** class.

filesystem

Description

The **filesystem** property defines the *FileSystemSync* (or the *FileSystem*) of the object and, by consequent, the **root** folder it belongs to. An object having a non null **filesystem** does not provide a way to access any object outside the filesystem **root** folder (sandboxing).

The **filesystem** property of an object is immutable. It is set when the object is created and is usually the *FileSystemSync* (or the *FileSystem*) of a parent object passed to the constructor.

fullPath

Description

The **fullPath** property returns the absolute path from the **root** folder of the filesystem to the entry.

isDirectory

Description

The **isDirectory** property returns **true** if the *EntrySync* object represents a directory, and **false** otherwise.

isFile

Description

The **isFile** property returns **true** if the *EntrySync* object represents a file, and **false** otherwise.

name

Description

The **name** property returns the name of the *EntrySync* object, excluding the path leading to it.

If you want to get the full path of an entry, you may consider using the **fullPath** property.

copyTo( )

EntrySync **copyTo**(DirectoryEntrySync *dest* [, String *name*])

Parameter	Type	Description
dest	DirectoryEntrySync	Directory to which copy the EntrySync
name	String	Name for the EntrySync copy
Returns	EntrySync	Reference of the moved entry

Description

The **copyTo( )** method copies the *EntrySync* object to a different location in the filesystem.

The EntrySync can be either a file entry or a directory entry. If you copy a file to a location where a file with the same name already exists, the method will try to delete and replace the existing file. If you copy a directory to a location where an empty directory with the same name already exists, the method will try to delete and replace the existing directory. Directory copies are recursive, that is, they copy all contents of the directory.

Pass in *dest* the destination directory where to copy the entry.

Optionally, you can pass in *name* a new name for the moved entry. If this parameter is omitted, by default the current EntrySync's name will be used.

The method returns a new *EntrySync* object corresponding to the moved entry.

An error will be generated if you try to:

- copy a directory inside itself or to any child at any depth;
- copy an entry into its parent if a *name* different from its current one is not provided;
- copy a file to a path occupied by a directory;
- copy a directory to a path occupied by a file;
- copy any element to a path occupied by a directory which is not empty.

getMetadata( )

Metadata **getMetadata**( )

Returns	Metadata	Information about the state of the entry

Description

The **getMetadata( )** method returns a *Metadata* object providing information about the state of a file or directory.

The returned *Metadata* object contains the following read-only properties:

Property	Type	Description
modificationTime	Date	Time at which the file or directory was last modified
size	Number	Size of the file in bytes (0 for directories)

getParent( )

DirectoryEntrySync **getParent**( )

Returns	DirectoryEntrySync	Parent directory of the entry

Description

The **getParent( )** method returns the parent *DirectoryEntrySync* of the EntrySync to which it is applied.
If the EntrySync is the root of its filesystem, the returned parent is the EntrySync itself.

moveTo( )

EntrySync **moveTo**(DirectoryEntrySync *dest* [, String *name*])

Parameter	Type	Description
dest	DirectoryEntrySync	Directory to which move the EntrySync
name	String	Name for the moved EntrySync
Returns	EntrySync	Reference of the moved entry

Description

The **moveTo( )** method moves the *EntrySync* object to a different location in the filesystem.
The EntrySync can be either a file entry or a directory entry. If you move a file to a location where a file with the same name already exists, the method will try to delete and replace the existing file. If you move a directory to a location where an empty directory with the same name already exists, the method will try to delete and replace the existing directory.
Pass in *dest* the destination directory where to move the entry.
Optionally, you can pass in *name* a new name for the moved entry. If this parameter is omitted, by default the current EntrySync's name will be used.
The method returns a new *EntrySync* object corresponding to the moved entry.
An error will be generated if you try to:

- move a directory inside itself or to any child at any depth;
- move an entry into its parent if a *name* different from its current one is not provided;
- move a file to a path occupied by a directory;
- move a directory to a path occupied by a file;
- move any element to a path occupied by a directory which is not empty.

remove( )

void **remove**( )

Description

The **remove( )** method deletes the entry (file or directory) from the filesystem.
An error is generated if:

- you try to delete a directory that is not empty,
- you try to delete the root directory of a filesystem.

toURL( )

String **toURL**( )

Returns	String	URL of the Entry

Description

The **toURL( )** method returns a URL that can be used to identify the *EntrySync*.
The returned URL describes a location on disk and is valid at least as long as that location exists.

File

The File class provides properties and methods that allow you to create and manipulate server-side *File* objects.

The basic way to create a *File* object is to execute the **File( )** method (from the Application class).

File objects contain references to disk files that may or may not actually exist on disk. For example, when you execute the **File( )** method to create a new file, a valid *File* object is created but nothing is actually stored on disk until you call the **create( )** method.
You should be aware that some methods will work correctly with the *File* object, even if the referenced file does not exist on disk. Some other methods require that the referenced file actually exists on disk. If you call any of them and the file does not exist, a "file not found" error is generated.

Blob Class Inheritance

In compliance with the W3C File API specification for HTML5, File class objects inherit from the **BLOB** class. It means that several additional useful features are available on *File* objects:

- To read the contents of a file as text, you can write:

```
var foo = File("c:/temp/myFile.txt");
foo.toString();
```

- You can use the **slice( )** method to access the bytes of a file.
- You can convert a *File* into a *Buffer* using **toBuffer( )** to access each byte of a file. Note however that since **toBuffer( )** makes a copy in memory of the file contents, you need to use **TextStream** or **BinaryStream** methods to modify the file.
- In general, you can pass a *File* object wherever Wakanda expects a *Blob*. See an example in the **sendChunkedData( )** method description.

creationDate

Description

The **creationDate** property returns the creation date for the *File* object.

This property can be modified.

exists

Description

The **exists** property returns *true* if the file referenced in the *File* object already exists at the defined path. Testing this property before creating or renaming files will help prevent errors.

Example

See the example for the **File( )** method.

extension

Description

The **extension** property returns the file name extension of the *File* object.

The extension string is returned without the "." character at the beginning.

The file referenced in the *File* object does not need to already exist on disk.

lastModifiedDate

Description

The **lastModifiedDate** property returns the last modification date for the *File* object.

This property can be modified.

name

Description

The **name** property gets or sets the short name of the *File* object without the path information. The file name is handled with its extension. If you do not want to manage the extension, use the **nameNoExt** property.

If you read the file name, the referenced file does not need to already exist on disk.
If you change the file name using this property, the referenced file must already exist. Note that it is renamed on disk but the **name**property is not updated in the *File* object.

Example

This property allows you to read or write the file name on disk:

```
var myFile = File ("c:/temp/invoices.txt");
var theName = myFile.name; // theName contains "invoices.txt"
myFile.name = theName + "_old"; // the file is renamed "invoices.txt_old" on disk
    //but myFile.name still contains "invoices.txt"
```

nameNoExt

Description

The **nameNoExt** property returns the short name of the *File* object without its path information or extension.

If you want to get the file name with its extension, use the **name** property.

parent

Description

The **parent** property returns a new *Folder* object containing the parent folder of the *File* or *Folder* object. The file or folder referenced in the object does not need to already exist on disk.

path

Description

The **path** property returns the full path of the *File* or *Folder* object, including the file or folder name itself.

The file or folder referenced in the object does not necessarily need to exist on disk. The returned path is expressed using the POSIX syntax (folders are separated with "/").

readOnly

Description

The **readOnly** property gets or sets the read-only status of the *File* object.

The property value is *true* if the referenced file is in read-only mode and *false* if it is in read/write mode.

The referenced file must already exist on disk and you must have the appropriate access rights to change its status.

size

Description

The **size** property returns the size of the *File* object expressed in bytes. The file referenced in the *File* object must already exist on disk when the property is read. Otherwise, an error is returned.

visible

Description

The **visible** property gets or sets the visibility status of the *File* or *Folder* object. The referenced file or folder must already exist on disk and, if you want to change the property value, you must have the appropriate access rights.

The property value is *true* if the referenced file or folder is visible, and *false* if it is not visible.

Example

This function returns **true** if all the files in a folder are visible:

```
function testAllFilesAreVisible(folder) {
    return folder.forEachFile(
        function (file) {
            return file.visible;
        }
    );
}
```

filesystem

Description

The **filesystem** property defines the *FileSystemSync* (or the *FileSystem*) of the object and, by consequent, the **root** folder it belongs to. An object having a non null **filesystem** does not provide a way to access any object outside the filesystem **root** folder (sandboxing).

The **filesystem** property of an object is immutable. It is set when the object is created and is usually the *FileSystemSync* (or the *FileSystem*) of a parent object passed to the constructor.

Example

```
File(folder, "untitled.txt") //will produce a File object with same filesystem as 'folder'.
File(FileSystemSync( "PROJECT"), "untitled.txt") // will produce a File object belonging to the "PROJECT" filesyster
File("c:/untitled.txt") // will produce a File object with a null filesystem
```

create( )

Boolean **create**( )

Returns	Boolean	True if the file has been created, otherwise False.

Description

The **create( )** method stores the file referenced in the *File* on disk. The path of the file has been defined in the **File( )** method (constructor).

The **create( )** method returns *True* if the file is created successfully. In all other cases, it returns *False*.

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 reference to the new data file
    var myDS = createDataStore(currentModel, dataFile);     // create and reference the datastore
}
```

Example

Wakanda allows you to combine a ***FileSystemSync*** and the path relative to this ***FileSystemSync*** in one single string.

If the following FileSystem is defined in the *fileSystem.json* file:

```
[
    {
            "name":"NOTES",
            "parent":"PROJECT",
            "path":"NotesFolder"
    }
]
```

You can write:

```
File("/NOTES/myText.txt");
// which is equivalent to:
File( FileSystemSync("NOTES"), "myText.txt")
```

getFreeSpace( )

Number **getFreeSpace**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Free space in bytes on the volume

Description

The **getFreeSpace( )** method returns the size of the free space (expressed in bytes) available on the volume where the ***File*** or ***Folder*** object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the ***File*** or ***Folder*** is stored, you can get the free space on the whole volume or only on your disk quota size, depending on the ***quotas*** parameter:

- If you pass *true* or the "NoQuotas" string in ***quotas*** (or omit the parameter), the method will take the whole volume into account. This is the action by default.
- If you pass *false* or the "WithQuotas" string in ***quotas***, the method will return only the free space of the disk quota.

getURL( )

String **getURL**([Boolean | String *encoding*])

Parameter	Type	Description
encoding	Boolean, String	true or "Encoded" = encode the URL, false or "Not Encoded" = do not encode the URL (default)
Returns	String	URL of the referenced file

Description

The **getURL( )** method returns the absolute URL of the ***File*** or ***Folder*** object.

By default, the returned URL characters are not encoded. You can set the encoding mode by using the ***encoding*** parameter:

- If you pass *true* or the "Encoded" string in ***encoding***, the URL will be encoded.
- If you pass *false* or the "Not Encoded" string in ***encoding*** (or omit the parameter), the URL is not encoded. This is the default action.

Example

The following example gets the URL from the path to a file:

```
var myfile = File ("c:/wk/web/img/logo.png");
var url = myfile.getURL(); // returns "file:///c:/wk/web/img/logo.png"
```

On Macintosh, the following example returns the URL to a file specific to Mac OS:

```
var myfile = File ("/Users/userName/Desktop/MySolution/MyProject/tmp/imageFile.jpg");
```

```
var url = myfile.getURL(); // returns "file:///Volumes/MacName/Users/userName/Desktop/MySolution/MyProject/tmp/image
```

getVolumeSize( )

Number **getVolumeSize**( )

Returns	Number	Size in bytes of the volume where the file is stored

Description

The **getVolumeSize( )** method returns the total size (expressed in bytes) of the volume where the ***File*** or ***Folder*** object is stored.

The referenced file or folder must exist on disk.

If the referenced file or folder is not found or if an error occurs, the method returns -1.

isFile( )

Boolean **isFile**(String *path*)

Parameter	Type	Description
path	String	POSIX absolute path
Returns	Boolean	True if path corresponds to an existing path, False otherwise

Description

The **isFile( )** class method can be used with the **File( )** constructor to know if *path* corresponds to a file on disk.

Pass in *path* a POSIX string containing an absolute path to a file on the disk. If the path corresponds to a file on the disk, **isFile( )** returns True. If the path corresponds to a non-existing file or a folder, **isFile( )** returns False.

Example

We want to know if "reports.proj" is a file:

```
var myFile = File.isFile("C:/wakanda/projects/reports.proj");
```

moveTo( )

void **moveTo**(File | String *destination* [, Boolean | String *overwrite*])

Parameter	Type	Description
destination	File, String	Destination file
overwrite	Boolean, String	True or "Overwrite" to override existing file if any, otherwise false or "KeepExisting"

Description

The **moveTo( )** method moves the file referenced in the ***File*** object (the source object) to the specified ***destination***. The file referenced in the ***File*** source object must already exist, otherwise the method returns a "File not found" error.

In the ***destination*** parameter, you can pass a ***File*** object or a string containing an absolute path or a URL to a file.

By default, a "File already exists" error will occur if there is a file with the same name as the source file at the defined ***destination***. You can change this behavior using the ***overwrite*** parameter:

- If you pass *true* or the "OverWrite" string in ***overwrite***, the method will delete and overwrite the existing file without any error.
- If you pass *false* or the "KeepExisting" string in ***overwrite*** (or omit the parameter), the existing file is left untouched and an error is generated. This is the default action.

Notes:

- You can use this method to rename a file on disk.
- Using this method, you can move a file from and to any folder on the same volume. If you want to move a file between two distinct volumes, use **copyTo( )** to “move” the file then delete the original copy of the file using the **remove( )** method.

Example

The following example renames a file in its own folder:

```
var myFile = new File ("c:/Documents/Invoice.txt") ; // get the File object and
myFile.moveTo ( "c:/Documents/Invoice_copy.txt") ; // rename it to the same location
```

next( )

Boolean **next**( )

Returns	Boolean	True if there is a next file in the iteration. Otherwise, it is false.

Description

The **next( )** method works with the file iterator: it puts the file pointer on the next file within an iteration of files, for example, in a **for** loop.

The method returns *true* if it has been executed correctly and *false* otherwise. More specifically, the method returns *false* when it reaches the end of a file collection, i.e., the files stored within a ***Folder*** object.

Example

See example for the **valid( )** method.

remove( )

Boolean **remove**( )

Returns	Boolean	True if the file was removed successfully. Otherwise, it returns false.

Description

The **remove( )** method removes the file or folder referenced in the ***File*** or ***Folder*** object from the storage volume. The file or folder referenced in the object must already exist on disk, otherwise this method returns a "File not found" error.
In the case of folders, this method deletes the folder as well as all of its contents.

The **remove( )** method returns *true* if the file or folder is removed successfully. In all other cases, it returns *false*.

setName ( )

Boolean **setName** (String *newName*)

Parameter	Type	Description
newName	String	New name for the disk file
Returns	Boolean	True if the file has been renamed successfully. Otherwise, it returns false.

Description

The **setName ( )** method allows you to rename a file on disk referenced in the ***File*** object. For this method to work, the file must already exist on disk and the application should have appropriate write permissions.

In the ***newName*** parameter, pass the new name of the file with its extension. The string must comply with the OS file naming rules.

If the method has been completed successfully, it returns *true*. If a problem occurred (file not found, no write permission, etc.), the method returns *false*.

Note that this method will rename the file on disk, but not the file name referenced in the ***File*** object.

valid( )

Boolean **valid**( )

Returns	Boolean	True if a current file reference exists. Otherwise, it returns false.

Description

The **valid( )** method works with the file iterator: it checks the validity of the pointer to the current ***File*** object within an iteration of files, for example, in a **for** loop.

The method returns *true* if the pointer is valid and *false* otherwise.

Example

In this example, we change all the file names in a folder (those at the top level of the folder) to uppercase letters:

```
var myFolder = Folder("c:/Wakanda/Files/");  // get a reference to the folder
for (var afile = myFolder.firstFile; afile.valid(); afile.next())
        // start at the first file and check whether it is valid in the folder
        // then move to the next file
    {
        afile.name = afile.name.toUpperCase();
    }
```

Example

See example for the **firstFile** property (Folder class).

FileEntrySync

A ***FileEntrySync*** object represents a file on a filesystem. This class inherits from the ***EntrySync*** class.

***Note:** The **createWriter( )** method is currently not supported in Wakanda.*

createWriter( )

FileWriterSync **createWriter**( )

Returns	FileWriterSync	New FileWriterSync associated with the file

Description

Note: This method is currently not implemented in Wakanda.

The **createWriter( )** method creates a new ***FileWriterSync*** associated with the file that the ***FileEntrySync*** represents.

A ***FileWriterSync*** object provides methods to write and modify files synchronously in a Worker. For more information about this object, please refer to the W3C File API: Writer specification.

file( )

File **file**( )

Returns	File	File referenced by the file entry

Description

The **file( )** method returns a ***File*** object that represents the current state of the file referenced by the ***FileEntrySync***.

The returned ***File*** object is a regular Wakanda object, that you can handle using the properties and methods of the **File**.

FileSystem Constructors

The following methods are constructors for *FileSystemSync* objects. They are available at the Wakanda Global object level (**Application** class).
The **resolveLocalFileSystemSyncURL( )** method returns a new *EntrySync* object based on a URL.

Example

In this example, we create a temporay file system to store a preferences file:

```
var fs = FileSystemSync(TEMPORARY,1024*1024); //create the filesystem

// Get the prefs directory, creating it if it doesn't exist
prefsDir = fs.root.getDirectory("preferences", {create: true});
// Create a tempo file, if and only if it doesn't exist.
try{
    prefFile = prefsDir.getFile("tempoPrefs.txt", {create: true, exclusive: true});
} catch (err) {
    // It already exists or something else went wrong
    // ... handle the exception
}
```

FileSystemSync( )

FileSystemSync **FileSystemSync**(Number | String *type* [, Number *size*])

Parameter	Type	Description
type	Number, String	0 = Temporary, 1 = Persistent, or name defined in the fileSystem.json file
size	Number	Expected storage space (in bytes)
Returns	FileSystemSync	New synchronous file system object

Description

The **FileSystemSync( )** method requests a *FileSystemSync* object referencing a sandboxed folder or file where application data can be stored. If the method is executed successfully, a new *FileSystemSync* object is returned.
Pass in *type* either a value describing the type of storage space you want the file system to reserve, or a string describing a FileSystem name defined in the *fileSystem.json* file (see **Using FileSystem Objects**):

- If you want to use a standard *FileSystemSync* object, 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. In Wakanda, this maps to the project Data folder. Thus, PERSISTENT is an alias for DATA.

  In this case, you need to pass in *size* a value indicating how much storage space, in bytes, the server expects to need.

- If you want to use a FileSystem defined in the *fileSystem.json* file, pass the "name" attribute value of the defined entry in *type*. In this case, the *size* value is ignored.

Example

Create a temporary file system at the project folder level which will be deleted after the JavaScript context is closed:

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

requestFileSystemSync( )

FileSystemSync **requestFileSystemSync**(Number | String *type* [, Number *size*])

Parameter	Type	Description
type	Number, String	0 = Temporary, 1 = Persistent, or name defined in the fileSystem.json file
size	Number	Expected storage space (in bytes)
Returns	FileSystemSync	New synchronous file system object

Description

The **requestFileSystemSync( )** method requests a *FileSystemSync* object referencing a sandboxed folder or file where application data can be stored. If the method is executed successfully, a new *FileSystemSync* object is returned.
Pass in *type* either a value describing the type of storage space you want the file system to reserve, or a string describing a FileSystem name defined in the *fileSystem.json* file (see **Using FileSystem Objects**):

- If you want to use a standard *FileSystemSync* object, 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. In Wakanda, this maps to the project Data folder. Thus, PERSISTENT is an alias for DATA.

  In this case, you need to pass in *size* a value indicating how much storage space, in bytes, the server expects to need.

- If you want to use a FileSystem defined in the *fileSystem.json* file, pass the "name" attribute value of the defined entry in *type*. In this case, the *size* value is ignored.

Example

Create a temporary file system at the project folder level which will be deleted after the JavaScript context is closed:

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

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.

FileSystemSync

The *FileSystemSync* object represents a server-side file system. Server-side file systems in Wakanda are based on the file system defined in the W3C File API specification. This concept was originally designed to provide access to sandboxed file systems allowing storage for Web applications (client side).

name

Description

The **name** property returns the name of the file system. The name of a file system is unique across the list of exposed file systems. It is automatically assigned by Wakanda when the file system is created.

root

Description

The **root** property returns the root directory of the file system. In Wakanda, the root directory is the project folder.
root.**fullPath** returns "/".

Folder

The Folder class provides properties and methods properties and methods that allow you to create and manipulate server-side *Folder* objects.

The basic way to create a *Folder* object is to execute the **Folder( )** method (available at the Global object level).

Folder objects contain references to folders that may or may not actually exist on disk. For example, when you execute the **Folder( )** method to create a new folder, a valid *Folder* object is created but nothing is actually stored on disk until you call the **create( )** method.
You should be aware of the fact that some methods will work correctly with a *Folder* object, even if the referenced folder does not exist on disk. Some other methods require that the referenced folder actually exists on disk: if they are called when the folder does not exist, a 'file not found' error is generated.

creationDate

Description

The **creationDate** property returns the creation date for the folder referenced in the *Folder* object.

This value can be modified.

Example

This example sorts a folder list by date:

```
function sortFolderListByCreationDate(folderA, folderB) {
    return (folderA.creationDate > folderB.creationDate) ? 1 : -1;
}
```

files

Description

The **files** property returns the list of the files located at the first level of the *Folder* object (files in subfolders are not returned). The property returns an array of *File* objects.

Example

This function logs all the files in a folder in alphabetical order:

```
     //Define the sort algorithm based on the file name
function sortArray(a, b) {
    if (a.name < b.name) return -1;
    if (a.name > b.name) return 1;
    return 0;
};

    //Sort and log the files in the folder
function logFiles(folder) {
    var files = folder.files;
    files.sort(sortArray);
    for(var i=0; i<files.length; i++)
        console.log(files[i].name);
};

var folder = new Folder("c:/wakanda/archives/");
logFiles(folder); //execute the function
```

firstFile

Description

The **firstFile** property returns a new *File* object referencing the file found in the first position of the *Folder* object. This property is useful when looping files in a folder.

If the folder does not contain any files, the property returns a *File* object with all the property values set to null.

Note: *The first file in a folder is selected internally by the file iterator; no specific order can be applied.*

Example

If you want to register each first-level file in a specific folder in an array to display a choice list, you can write the following:

```
var folder = Folder("c:/wakanda/");
var result2 = [];
var fileIter = folder.firstFile;  // initialize the iteration
while (fileIter.valid())  // while there are files to process
{
    result2.push(fileIter.path);
    fileIter.next();  // increment the iteration
}
result2;   // display the object in the JavaScript Editor
```

Note: *The result of this example is exactly the same as the result of example 1 for the* ***forEachFile( )*** *method. The only advantage of this example is that it allows you to add tests to stop the iteration at any time.*

firstFolder

Description

The **firstFolder** property returns a new *Folder* object containing the subfolder found in the first position of the *Folder* object. This property is useful when you loop the subfolders in a folder.

If the folder does not contain any subfolders, the property returns a *Folder* object with all the property values set to null.

Note: *The first subfolder in a folder is selected internally by the iterator; no specific order can be applied.*

folders

Description

The **folders** property returns the list of the subfolders located at the first level of the *Folder* object. The property returns an array of *Folder* objects.

modificationDate

Description

The **modificationDate** property returns the last modification date for the folder referenced in the *Folder* object.
This value can be modified.

name

Description

The **name** property gets or sets the name of the *Folder* object without the path information.
When you read this property, the referenced folder does not need to already exist on disk.
If you change the folder name using this property, the referenced folder must already exist. Note that it is renamed on disk but the **name** property is not updated in the *Folder* object.

Example

This function returns 1 if the name of folderA is greater than folderB in alphabetical order. Otherwise, it returns -1.

```
function sortFolderListByName(folderA, folderB) {
    return (folderA.name > folderB.name) ? 1 : -1;
}
```

parent

Description

The **parent** property returns a new *Folder* object containing the parent folder of the *File* or *Folder* object. The file or folder referenced in the object does not need to already exist on disk.

path

Description

The **path** property returns the full path of the *File* or *Folder* object, including the file or folder name itself.
The file or folder referenced in the object does not necessarily need to exist on disk. The returned path is expressed using the POSIX syntax (folders are separated with "/").

visible

Description

The **visible** property gets or sets the visibility status of the *File* or *Folder* object. The referenced file or folder must already exist on disk and, if you want to change the property value, you must have the appropriate access rights.
The property value is *true* if the referenced file or folder is visible, and *false* if it is not visible.

nameNoExt

Description

The **nameNoExt** property returns the name of the *Folder* object without path information and without the main extension (if any).
If you want to get the full name of the folder with its extension, use the **name** property.

extension

Description

The **extension** property returns the folder name extension of the *Folder* object.
If the folder name does not have an extension, the property value is *undefined*.

exists

Description

The **exists** property returns *true* if the folder referenced in the *Folder* object already exists at the defined path.

filesystem

Description

The **filesystem** property defines the *FileSystemSync* (or the *FileSystem*) of the object and, by consequent, the **root** folder it belongs to. An object having a non null **filesystem** does not provide a way to access any object outside the filesystem **root** folder (sandboxing).
The **filesystem** property of an object is immutable. It is set when the object is created and is usually the *FileSystemSync* (or the *FileSystem*) of a parent object passed to the constructor.

Example

See examples for the **filesystem** property.

create( )

Boolean **create**

Returns	Boolean	True if the folder has been created. Otherwise, it returns false.

Description

The **create( )** method creates the folder referenced in the *Folder* object on disk. The path of the folder is defined in the **Folder( )** method.

The **create( )** method returns *true* if the file is created successfully. Otherwise, it returns *false*.

If a folder already exists at the defined path (with or without any contents), the method does nothing but returns *true*. If the path contains a hierarchy of non-existing subfolders, they are created if necessary.

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();
```

forEachFile( )

void **forEachFile**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to invoke for each file in the folder
thisArg	Object	Token object that will be bound to 'this' in the handler

Description

***Note:** You can also call this method using its alias **each( )**.*

The **forEachFile( )** method executes the *callbackFn* function once for each file present at the first level of the *Folder* object.

Only first level files are taken into account (files in subfolders are left untouched). If you want to process all the files in a folder at every level of recursivity, you may want to consider using the **parse( )** method.

The files are processed in no particular order (depending on the file system). If the file order is important for a specific processing, you may want to use the **files** property and sort the array:

```
var files = myfolder.files;
files.sort();
for(...) // iterate on 'files' array
```

The *callbackFn* function accepts three parameters: *theFile*, *iterator*, and *theFolder*.

- The first parameter, ***theFile***, is the file currently being processed. When the function is executed, this parameter receives the file on which it is iterating. You can then perform any type of operation on the file.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the file currently being processed in the folder. You can use it, for example, to display a counter.
- The third (optional) parameter, *theFolder*, receives the parent *Folder* being processed.

If a *thisArg* parameter is provided, it will be used as the **this** value each time the *callbackFn* is called. If it is not provided, **undefined** is used instead.

If existing files in the folder are modified, their values as passed to the callback will be the value when **forEachFile( )** visits them. Files that are deleted after the call to **forEachFile( )** begins and before being visited are not visited.

Example

In this example, we register each first level file in a specific folder in an array, for example to display a choice list by using the **forEachFile( )** method:

```
var folder = Folder("c:/wakanda/");
 var result = [];
 folder.forEachFile(function(file)
 {
     result.push(file.path); // store the file path
 });
 result; // display the object in the JavaScript Editor
```

***Note:** The result for this example is exactly the same as the example of the **firstFile** method. The **forEachFile( )** method simplifies the iteration.*

Example

In this example, fill an array with all the files matching a specific criteria in the 'Wakanda' folder.

```
var list = [];
var myFolder = Folder('c:/Wakanda/');
myFolder.forEachFile(
    function (file) {
        if ((file.lastModifiedDate+ this.ms) > +(new Date()))
        {
            list.push(file);
        }
    },
    {ms: 42} // thisArg parameter
);
```

forEachFolder( )

void **forEachFolder**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to call for each subfolder
thisArg	Object	Token object that is bound to 'this' in the handler

Description

The **forEachFolder( )** method executes the ***callbackFn*** function once for each subfolder present at the first level of the ***Folder*** object. The subfolders are processed in no particular order (it depends on the file system).

Only first level subfolders are taken into account (nested subfolders are left untouched).

The *callbackFn* function accepts three parameters: *theSubfolder*, *iterator*, and *theFolder*.

- The first parameter, ***theSubfolder***, is the subfolder currently being processed. When the function is executed, this parameter receives the folder on which it iterates. You can then perform any type of operation on the subfolder.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the subfolder currently being processed in the parent folder. You can use it, for example, to display a counter.
- The third (optional) parameter, ***theFolder***, receives the parent ***Folder*** being processed.

If a ***thisArg*** parameter is provided, it will be used as the **this** value each time the ***callbackFn*** is called. If it is not provided, **undefined** is used instead.

If existing subfolders in the folder are modified, their values as passed to callback will be the values at the time **forEachFolder( )** visits them. Folders that are deleted after the call to **forEachFolder( )** begins and before being visited are not visited.

getFreeSpace( )

Number **getFreeSpace**([Boolean | String *quotas*])

Parameter	Type	Description
quotas	Boolean, String	true or "NoQuotas" = consider the whole volume (default), false or "WithQuotas" = consider only the allowed size for the quota
Returns	Number	Free space in bytes on the volume where the folder is stored

Description

The **getFreeSpace( )** method returns the size of the free space (expressed in bytes) available on the volume where the ***File*** or ***Folder*** object is stored. The referenced file or folder must exist on disk.

If system disk quotas have been activated on the volume where the ***File*** or ***Folder*** is stored, you can get the free space on the whole volume or only on your disk quota size, depending on the ***quotas*** parameter:

- If you pass *true* or the "NoQuotas" string in ***quotas*** (or omit the parameter), the method will take the whole volume into account. This is the action by default.
- If you pass *false* or the "WithQuotas" string in ***quotas***, the method will return only the free space of the disk quota.

getURL( )

String **getURL**([Boolean | String *encoding*])

Parameter	Type	Description
encoding	Boolean, String	true or "Encoded" = encode the URL, false or "Not Encoded" = do not encode the URL (default)
Returns	String	URL of the referenced folder

Description

The **getURL( )** method returns the absolute URL of the ***File*** or ***Folder*** object.

By default, the returned URL characters are not encoded. You can set the encoding mode by using the ***encoding*** parameter:

- If you pass *true* or the "Encoded" string in ***encoding***, the URL will be encoded.
- If you pass *false* or the "Not Encoded" string in ***encoding*** (or omit the parameter), the URL is not encoded. This is the default action.

Example

The example below returns the URL from the path of a folder:

```
var myfolder = Folder("c:/wk/web/img/");
var url = myfolder.getURL(); // will return "file:///c:/wk/web/img/"
```

getVolumeSize( )

Number **getVolumeSize**( )

Returns	Number	Size in bytes of the volume where the folder is stored

Description

The **getVolumeSize( )** method returns the total size (expressed in bytes) of the volume where the ***File*** or ***Folder*** object is stored.

The referenced file or folder must exist on disk.

If the referenced file or folder is not found or if an error occurs, the method returns -1.

isFolder( )

Boolean **isFolder**(String *path*)

Parameter	Type	Description
path	String	POSIX absolute path
Returns	Boolean	True if the path corresponds to a folder, False otherwise

Description

The **isFolder( )** class method can be used with the **Folder( )** constructor to know if *path* corresponds to a folder on disk.

Pass in *path* a POSIX string containing an absolute path to a folder on the disk. If the path corresponds to a folder, **isFolder( )** returns True. If the path corresponds to a non-existing folder or a file, **isFolder( )** returns False.

Example

We want to know if "reports.proj" is a folder:

```
var myFolder = Folder.isFolder("C:/wakanda/projects/reports.proj");
```

next( )

Boolean **next**

Returns	Boolean	True if there is a next subfolder in the iteration. Otherwise, it returns false.

Description

The **next( )** method works with the subfolder iterator. It puts the folder pointer on the next subfolder in an iteration of subfolders, for example in a **for** loop.

The method returns *true* if it has been executed correctly and *false* otherwise. This method returns *false* when it reaches the end of a folder collection, i.e., the subfolders stored in a ***Folder*** object.

parse( )

void **parse**(Function *callbackFn* [, Object *thisArg*])

Parameter	Type	Description
callbackFn	Function	Handler function to call for each file and subfolder
thisArg	Object	Token object that will be bound to 'this' in the handler

Description

The **parse( )** method executes the ***callbackFn*** function once for each file or subfolder present in the ***Folder*** object. This method also executes the ***callbackFn*** function on subfolders in the subfolders recursively.

The files or subfolders are processed in no particular order (depending on the file system).

The ***callbackFn*** function accepts three parameters: *item*, *iterator*, and *theFolder*.

- The first parameter, ***item***, represents the file currently being processed. When the function is executed, this parameter receives the file on which it iterates. You can then perform any type of operation on the current item.
- The second (optional) parameter, *iterator*, is the iterator. When the function is executed, this parameter receives the position of the file currently being processed in the parent folder. You can use it, for example, to display a counter.
- The third (optional) parameter, ***theFolder***, receives the parent ***Folder*** being processed.

If a ***thisArg*** parameter is provided, it will be used as the **this** value each time the ***callbackFn*** is called. If it is not provided, **undefined** is used instead.

If existing elements (files or subfolders) of the folder are modified, their values as passed to callback will be the values at the time **parse( )** visits them. Items that are deleted after the call to **parse( )** begins and before being visited are not visited.

Example

In a "Messages" folder containing several text files stored in several subfolders, this example loads and concatenates the contents of each file into a single text variable. It also handles errors that may occur.

```
var folder = Folder("c:/Messages/");
var result = [];
var textResult = "";
folder.parse(function(file, position) // for each file in the folder
{
    var ext = file.extension.toLowerCase(); // get .TXT or.Text
    if (ext == "text" || ext == "txt")
    {
        result.push(file.path); // put the file in an array
        try  // error handling block
        {
            var text = loadText(file);  // we already know that files are of type Text
            if (text != null)  // do not take empty files
                textResult += "pos# "+position+" : "+text + "\n";  // write the position and its contents
        }
        catch (err)
        {
```

```
            textResult += "*** " + file.path+" could not be loaded *** \n";
        }
    }
});
textResult; // display the text variable in the JavaScript Editor
```

Example

In this example, we count the files with the read-only and read/write status in a folder (at all sublevels):

```
var folder = Folder("c:/wakanda/");
var result = { readOnlyCount: 0, readWriteCount: 0 } // initialize the count to zero
folder.parse(function(file) // retrieve the 'file' parameter
{
    if (file.readOnly)
        result.readOnlyCount++;
    else
        result.readWriteCount++;
});
result; // display the resulting object in the JavaScript Editor
```

remove( )

Boolean **remove**( )

Returns	Boolean	True if the folder was removed successfully. Otherwise, it returns false.

Description

The **remove( )** method removes the file or folder referenced in the ***File*** or ***Folder*** object from the storage volume. The file or folder referenced in the object must already exist on disk, otherwise this method returns a "File not found" error.
In the case of folders, this method deletes the folder as well as all of its contents.

The **remove( )** method returns *true* if the file or folder is removed successfully. In all other cases, it returns *false*.

removeContent( )

Boolean **removeContent**( )

Returns	Boolean	True if the folder's contents were removed successfully. Otherwise, it returns false.

Description

The **removeContent( )** method removes the contents of the folder referenced in the ***Folder*** object from the storage volume. The folder referenced in the object must already exist on disk, otherwise the method returns a "File not found" error.
Only the contents of the folder is removed (files and subfolders), but the folder itself is left untouched.

The **removeContent( )** method returns *true* if the folder's content are successfully removed. Otherwise, it returns *false*.

setName( )

void **setName**(String *newName*)

Parameter	Type	Description
newName	String	New name for the folder on disk

Description

The **setName( )** method allows you to rename the folder referenced in the ***Folder*** object on disk. For the method to work, the folder must already exist on disk and the application should have appropriate write permissions to do so.

In the *newName* parameter, pass the folder's new name. The string must comply with the OS file naming rules.

If the method was successful, it returns *true*. If a problem occurred (a folder with the same name already exists, no write permission, etc.), the method returns *false*.

Note that this method will rename the folder on the disk, but not the folder name referenced in the ***Folder*** object.

valid( )

Boolean **valid**( )

Returns	Boolean	True if a current folder exists. Otherwise, it returns false.

Description

The **valid( )** method works with the subfolder iterator. It checks the validity of the pointer to the current folder within an iteration of folders, for example in a **for** loop.

This method returns *true* if the pointer is valid and *false* if it is not.

TextStream

TextStream class methods allow you to handle and parse text-based streams mapped to disk files. Unlike the **BinaryStream** methods, TextStream methods support a *charSet* parameter for encoding/decoding streams.

To create a *TextStream* object, you need to execute the **TextStream( )** constructor method (available at the Global object level).

Note: *If you need to load all the contents of a text file at once, you might also consider using the* ***loadText( )*** *from the Application class.*

close( )

void **close**

Description

The **close( )** method closes the file referenced in the *TextStream* object.

The referenced file is opened when you execute the **TextStream( )** method and stays open until you call **close( )**.

end( )

Boolean **end**( )

Returns	Boolean	true when the cursor is beyond the last character of the file (in buffer)

Description

The **end( )** method returns *true* if the cursor position is after the last character of the file referenced in the *TextStream* object.

As long as the cursor is located within the file, the **end( )** method returns *false*.

This method is useful when you want to read a large file. For example, you can read a file line by line (by using the **read( )** method with an empty string) until **end( )** returns *true*.

flush( )

void **flush**( )

Description

The **flush( )** method saves the contents of the buffer to the disk file referenced in the *TextStream* object.

When you execute several method calls to write data into a *TextStream* object, for optimization reasons the data is stored in a buffer that is saved to disk when the stream is closed. This method allows you to save the buffer at any time during the process without having to close the stream.

getPos( )

Number **getPos**( )

Returns	Number	Position of the cursor in the stream

Description

The **getPos( )** method returns the current position of the cursor in the *TextStream* object, that is, the number of characters read or written from the beginning of the object.

getSize( )

Number **getSize**( )

Returns	Number	Current size of the stream (in bytes)

Description

The **getSize( )** method returns the current size of the stream.

When you are reading the stream, this method returns the size of the file referenced in the *TextStream* object.

When you are writing the stream, this method returns the current size of the stream in memory.

read( )

String **read**([Number | String *numBytesOrDelimiter*])

Parameter	Type	Description
numBytesOrDelimiter	Number, String	Number of bytes to read or Character at which to stop reading
Returns	String	Received data

Description

The **read( )** method reads characters from the file referenced in the *TextStream* object. Whatever the character set defined in the *TextStream* object, the **read( )** method returns text expressed in UTF-16 (characters are converted if necessary).

The characters read are returned as a string value.

- To read a particular number of characters, pass this number in *numBytesOrDelimiter*. This number of characters will be returned and the internal cursor position will be updated. You can read up to 2GB of text.
  Note: *The value you pass actually represents the number of bytes. Characters are usually encoded on one byte, but certain characters are encoded on two bytes, like accented characters. The correspondence between the number of characters/number of bytes may vary.*
- To read the whole contents of the file, pass 0 in *numBytesOrDelimiter* or omit the parameter.
- To read data until a particular character is encountered, pass that character in *numBytesOrDelimiter*. This method will return all the characters in the

file from the current cursor position until it finds the delimiter string (which is not returned). In this case, if the delimiter string is not found, the **read( )** method will read to the end of the file.

If you pass an empty string ("") in *numBytesOrDelimiter*, this method will return one line of text -- it returns all the characters until it finds an end-of-line character (*carriage return* or *line feed*).

When reading a file, the first **read( )** method begins at the beginning of the file. Reading subsequent data begins at the character following the last byte read.

When attempting to read past the end of the file, **read( )** will return the data read up to that point and the **end( )** method will return *true*. Then, the next **read( )** will return an empty string.

rewind( )

void **rewind**( )

Description

The **rewind( )** method moves the stream cursor to the beginning of the *TextStream* object.

After this method is executed, the **getPos( )** method returns 0.

setPos( )

void **setPos**(Number *offset*)

Parameter	Type	Description
offset	Number	New cursor position in the stream

Description

The **setPos( )** method moves the stream cursor to the *offset* position in the *TextStream* object.

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, and "Overwrite" to replace the file with new data.
charSet	Number	Character set of the text. By default, the value is 7, which is UTF-8.
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)

***Warning**: The charSet parameter (if set) and the actual file charset must match, otherwise the character decoding operation could cause errors.*

Example

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

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

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

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

        logFile: null
    }

    log.set(file);

    return log;
}
```

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

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

Example

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

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

```
    }
while(mystream.end()==false)
mystream.close();
```

write( )

void **write**(String *text*)

Parameter	Type	Description
text	String	Data of type Text to write in the stream

Description

The **write( )** method writes the data you passed in the ***text*** parameter in the ***TextStream*** object. The text is written at the current position of the cursor in the stream. The cursor position is updated afterwards.