Syncplify.me AFT! Manual

Table of contents

Welcome

Thank you for choosing Syncplify.me AFT!, the perfect solution to create your own, flexible, automated, managed file transfer tasks.

With Syncplify.me AFT! you can:

- create source-code AFT tasks in aftJS which is an extended version of JavaScript specifically designed by Syncplify for managed file transfer
- create visual AFT tasks by simply assembling "building blocks", to bring the power of AFT to people who are not familiar with computer programming
- run your AFT tasks in various ways, including:
    - invoking tasks from anywhere via our built-in REST API
    - running tasks from the command-line
    - running tasks by simply double clicking on script files
    - scheduling tasks using the built-in scheduler
    - scheduling tasks using your operating system's native scheduler/cron
- log every event in JSON-formatted log files for easy import and analysis in the most widespread log analyzers (ex: LogRhythm, ManageEngine, SumoLogic, Loggly, ...)
- and so so much more...

Getting Started

Getting help

This Manual is the primary source of help and learning for Syncplify.me AFT!, but we also have hundreds of articles in our online Knowledge Base. We strongly recommend to check/search the Knowledge Base every time you have a residual doubt after reading the manual: chances are that the answer to your question is already there waiting for you.

If the Manual and the Knowledge Base were still not enough, you can open a support ticket here.
Tickets can be opened in 2 categories:

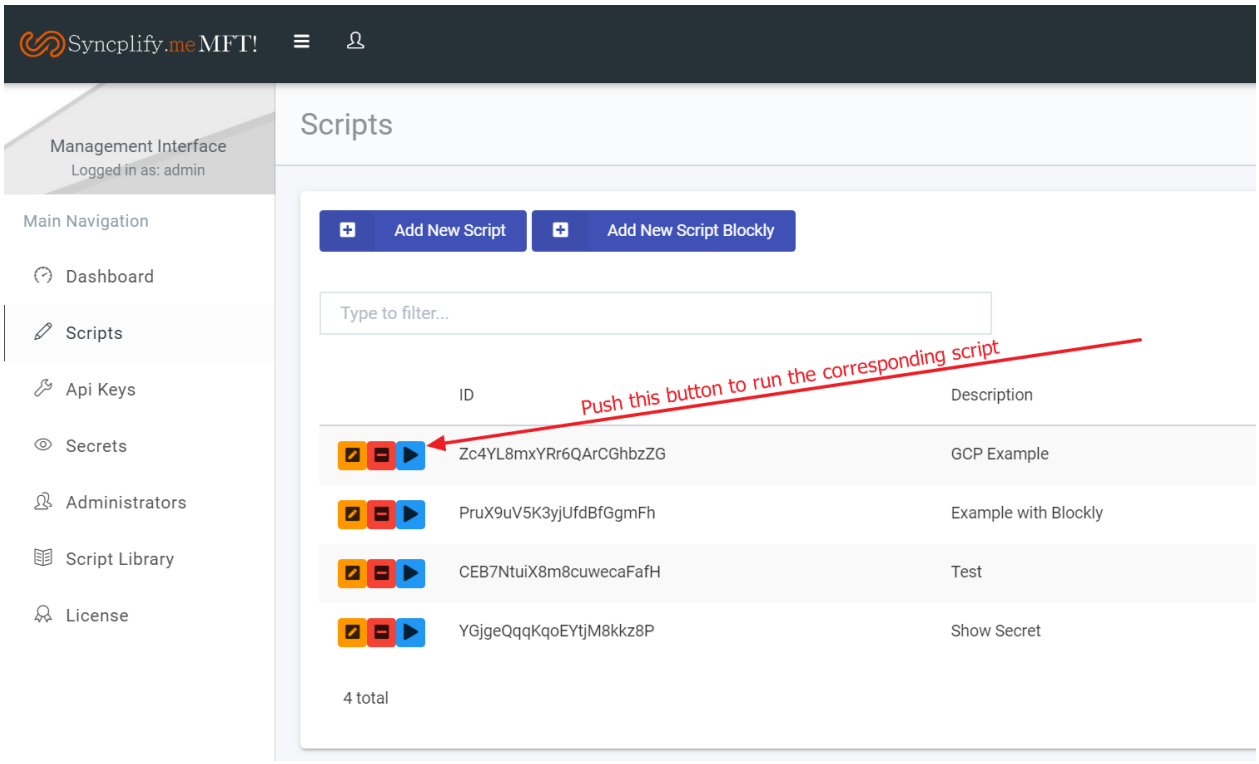
1. to receive help and support, if you are an existing customer
2. to ask pre-sales questions, if you're planning on buying our product

How to run your aftJS scripts

Syncplify.me AFT! is very flexible, and allows you to run your scripts (AFT tasks) in several different ways, so that you can run them interactively, or even automate their execution when necessary.

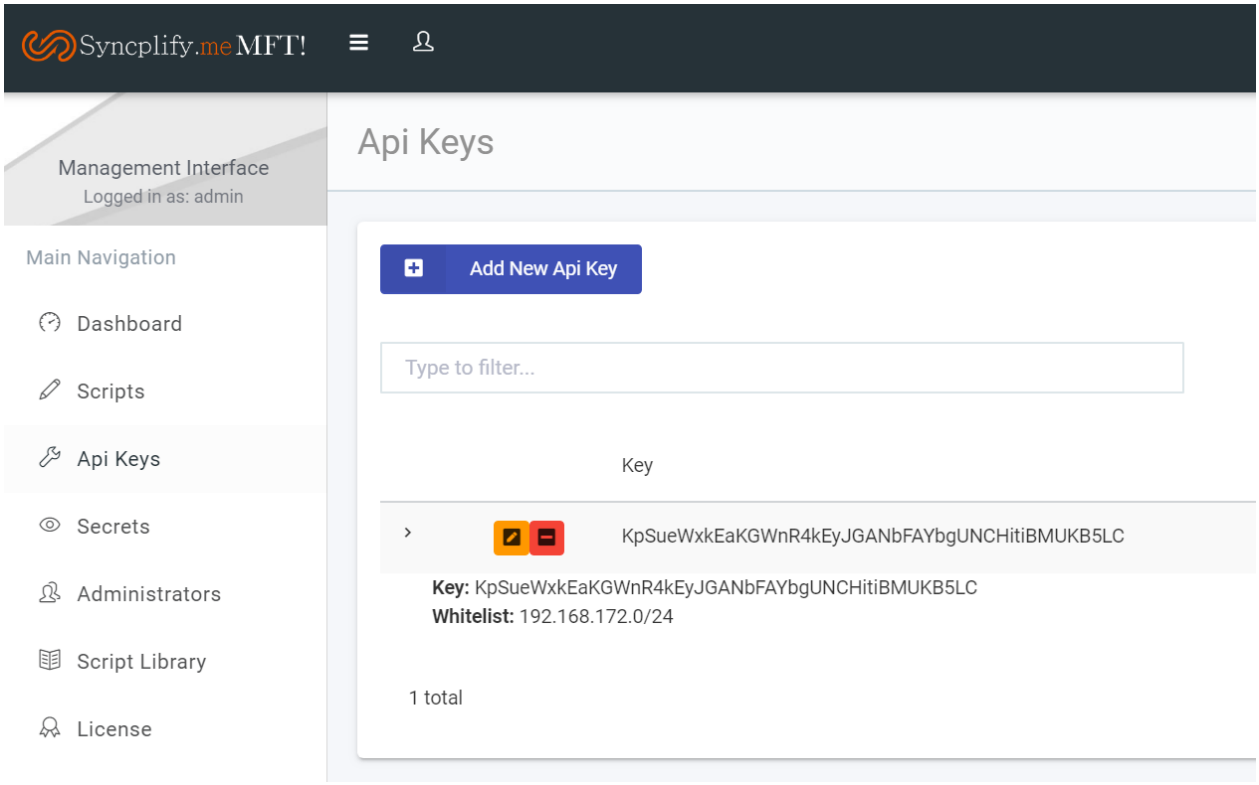
1. Run scripts from within the web interface

This is probably the most intuitive way to run your scripts. In the "Scripts" section of the web interface, simply hit the "run" button next to the script you want to execute.



2. Run scripts via REST API

Syncplify.me AFT! also allows you to run your scripts via REST API. To do so, first you have to create an API Key, as shown in the picture here below.



Please keep in mind that it's highly recommended to limit each API Key you create with a "whitelist". A whitelist is a list of IP addresses and networks from which such API Key will be accepted. If someone tries to run a script using this API Key from a machine (remote IP address) that's not in the Key's whitelist, the

script will not be executed. This provides an additional level of security and safety.

Once your API Key exists, then you can run your scripts by simply consuming the following REST API endpoint (which is super easy to do in practically every modern development language and/or shell):

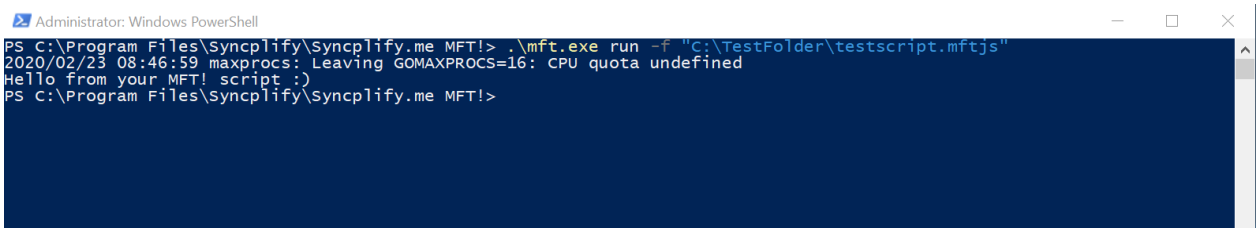
```
curl --request POST \
  --url https://127.0.0.1:44399/v1/jobs \
  --header 'authorization: Bearer ' \
  --header 'content-type: application/json' \
  --header 'x-api-key: KpSueWxkEaKGWnR4kEyJGANbFAYbgUNCHitiBMUKB5LC' \
  --data '{
  "jobType": "SCRIPT",
  "id":"qHULASriEP9Hq7fRRsB4Jj",
  "params": [
    {
      "name": "firstParam",
      "value": "Mickey Mouse"
    },
    {
      "name": "somethingElse",
      "value": "Goofy"
    }
  ]
}'
```

Your API Key

ID of the script you wish to run

3. Run scripts interactively from the command prompt

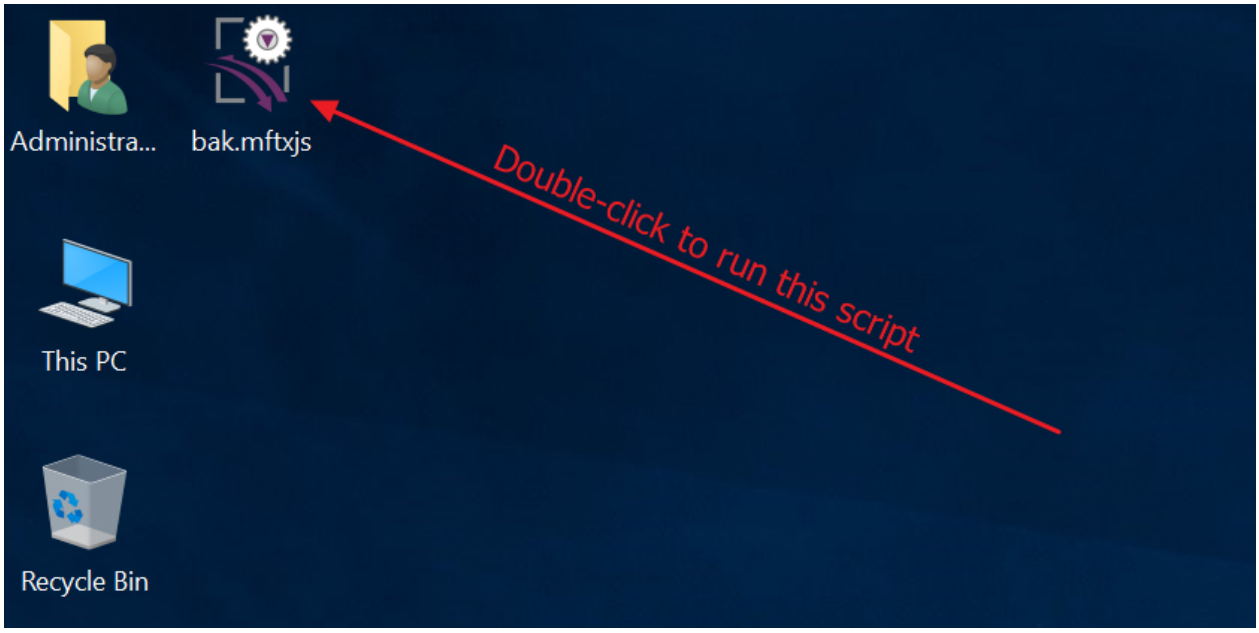
From the command prompt (or even from the PowerShell) you can invoke the execution of a script that's saved in a file very easily by using the "run" command built into AFT! itself, and the --file (or -f shorthand) parameter, as shown in the picture here below.



This method is very useful also to schedule the execution of your scripts (saved as files) from the Windows Scheduler (or cron in Linux).

4. Run scripts interactively by double-clicking on them

Since you can save scripts as normal plain text files, you can also run them by simply double-clicking on them from within your operating system's window manager (Explorer in Windows, or GNOME/KDE/... in Linux).



Note: in Windows, if your file has a ".aftjs" extension it will be run and then the console it's been run into will remain open for you to check the resulting log; if, on the other hand, the file has a ".aftxjs" extension, the console window will be automatically closed after the script finishes running.

Environment variables

Most aspects of Syncplify.me AFT! can be managed and/or initialized by means of environment variables. This is a "best practice" because environment variables are safe (secured by the operating system), don't require to store values inside the executable's memory, and are compatible and supported by all operating system, in all physical and virtual environments, including containers.

Here's a list of environment variables supported by Syncplify.me AFT!, together with their explanation:

SMAFT_EK: typically initialized by the software installer, this is a random string from which AFT! derives the encryption key to encrypt its own configuration files. It's important to understand that if you need to move an instance of AFT! from one machine to another, you'll also need to make sure both machines have this environment variable set to the exact same string value, otherwise the destination machine won't be able to read/access its configuration, and the software will not work.

SMAFT_BINDTO: this is the IP address AFT!'s HTTP server should bind to upon starting. If you don't specify it, AFT! will bind to **127.0.0.1** by default, and therefore it will be only accessible from **localhost**.

SMAFT_PORT: this is the TCP port (1-65535) AFT!'s HTTP server will bind to upon starting. If you don't specify it, AFT! will bind to port **44399**.

SMAFT_CERT: this is the fully-qualified path to an X.509 certificate bundle file. This variable is optional. If not defined, AFT! will use the default path, which is **%COMMON_APP_DATA%\Syncplify.me\AFTv1\certs\server.crt** on Windows, or **%XDG_DATA_DIRS[0]%/Syncplify.me/AFTv1/certs/server.crt** on Linux/POSIX.

SMAFT_KEY: this is the fully-qualified path to an X.509 certificate private key file. This variable is optional. If not defined, AFT! will use the default path, which is **%COMMON_APP_DATA%\Syncplify.me\AFTv1\certs\server.key** on Windows, or **%XDG_DATA_DIRS[0]%/Syncplify.me/AFTv1/certs/server.key** on Linux/POSIX.

SMAFT_BODYLIMIT: this is the maximum incoming request size that AFT!'s built-in HTTP server will accept; if not specified, the default value is **32M**. This is a string value, composed of a numerical prefix followed by a letter indicating the unit of measurement. Example: 32M means 32 MegaBytes. Allowed post-fix letters are K for KiloBytes, M for MegaBytes, G for GigaBytes, T for TeraBytes, and P for

PetaBytes.

SMAFT_RUNNERS: this is the maximum number of jobs/tasks that AFT! will execute concurrently... if you start more than SMAFT_RUNNERS tasks, all tasks exceeding this number will be queued, and executed later when one of the running tasks terminates and frees up system resources. This is an optional setting, and its default value is **32**. Please, also be aware of the fact that the free/evaluation edition of AFT! forces this number to be always **1 (one)** regardless of what you set in this environment variable.

The aftJS language

The aftJS language is like **JavaScript**, actually it is JavaScript, nearly 100% compatible with the **ECMA5 specification**, including strict mode and regular expressions, and with **some ECMA6 functionality** as well. Furthermore, and more importantly, it has several additional functions and methods specifically designed to develop Managed File Transfer scripts.

Caveats:

The only known caveat at this time is that WeakMap maintains "hard" references to its values. This means if a value references a key in a WeakMap or a WeakMap itself, it will not be garbage-collected until the WeakMap becomes unreferenced. To illustrate this, see the following script:

```
{
  var m = new WeakMap();
  var key = {};
  m.set(key, {key: key});
  // or m.set(key, key);
  key = undefined; // The value will NOT become garbage-collectable at this
point
  m = undefined; // But it will at this point
}
```

System requirements

Supported operating systems

- **Windows**: all versions from XP through Windows 10, including Windows Server and all R2 versions
- **Linux**: tested on Ubuntu, CentOS, RedHat, Debian (should work on other distributions as well) on both x86 and Arm architectures
- **MacOSX**: coming soon
- **Containers**: Docker image coming soon

Minimum hardware requirements

- **RAM**: 512 MB
- **Free disk space**: 40 MB
- **CPU cores**: 1+

Options (for ALL client objects)

```
Options = {
 StopOnTransferError       // boolean
 DownloadPolicy            // enum: NeverOverwrite, AlwaysOverwrite,
OverwriteIfDiffSize, OverwriteIfNewer
 UploadPolicy              // enum: NeverOverwrite, AlwaysOverwrite,
OverwriteIfDiffSize, OverwriteIfNewer
 OnDownloadGrantTo         // string (username of a user to whom AFT! will
grant access to downloaded file(s)
 AdjustTimeOnDownload      // bool
```

```
 AdjustTimeOnUpload         // bool
 DownloadWithTempName       // bool
 UploadWithTempName         // bool
 DeleteSourceAfterDownload  // bool
 DeleteSourceAfterUpload    // bool
 VersionedDownload          // bool
 VersionedUpload            // bool
 VersionsToKeepLocal        // integer - default: 3
 VersionsToKeepRemote       // integer - default: 3
 OTFE                       // bool
 OTFEKey                    // string
 OTFEKeyFromSecret          // string
}
```

Every client object, regardless of the file transfer protocol it implements, will have an`Options` property like the one described here above.
Here below you can find an explanation of what each one of such options means.

`StopOnTransferError`: if **`true`**, any upload or download operation will immediately terminate if a file transfer error occurs, otherwise (if **`false`**) Syncplify.me AFT! will try to keep uploading/downloading the remaining queued files; this property defaults to **`false`**.

`DownloadPolicy` is the policy that the client object will apply to all downloads, and specifically:

- `NeverOverwrite`: if a file with the same name already exists on the local file system, it will not be downloaded (this is the default value)
- `AlwaysOverwrite`: all files will always be downloaded, even if it means that local files are going to be overwritten
- `OverwriteIfDiffSize`: if a local file with the same name exists, the remote file will be downloaded (overwriting the local one) only if the size is different
- `OverwriteIfNewer`: if a local file with the same name exists, the remote file will be downloaded (overwriting the local one) only if the remote file is more recent (looking at the last modification date)

`UploadPolicy` is the policy that the client object will apply to all uploads, and specifically:

- `NeverOverwrite`: if a file with the same name already exists on the remote file server, it will not be uploaded (this is the default value)
- `AlwaysOverwrite`: all files will always be uploaded, even if it means that remote files are going to be overwritten
- `OverwriteIfDiffSize`: if a remote file with the same name exists, the local file will be uploaded (overwriting the remote one) only if the size is different
- `OverwriteIfNewer`: if a remote file with the same name exists, the local file will be uploaded (overwriting the remote one) only if the local file is more recent (looking at the last modification date)

`OnDownloadGrantTo`: when AFT! is running as a system service, all downloaded files will be owned by "System" (on Windows) or "root" (on Linux/POSIX systems). If you specify this option, then AFT! will grant access (in Windows) or ownership (in Linux/POSIX) to all downloaded files to the**username** specified by this option.

`AdjustTimeOnDownload`: if **`true`**, the last modification time of each downloaded file will be adjusted to match the last modification time of the original file on the remote file server; this property defaults to**`true`**.

`AdjustTimeOnUpload`: if **`true`**, the last modification time of each uploaded file will be adjusted to match the last modification time of the original file on the local file system; this property defaults to**`true`**.

`DownloadWithTempName`: if **`true`**, all downloads will be operated using a temporary file, and only after a successful and complete download the temporary file will be renamed to the actual original name of the downloaded file. This property defaults to **`false`**.

`UploadWithTempName`: if **`true`**, all uploads will be operated using a temporary file, and only after a successful and complete upload the temporary file will be renamed to the actual original name of the

uploaded file. This property defaults to **false**.

DeleteSourceAfterDownload: if **true**, after a file has been successfully downloaded, the original file (on the remote file server) is deleted, effectively turning the download into a "file move" operation from the remote to the local side.

DeleteSourceAfterUpload: if **true**, after a file has been successfully uploaded, the original file (on the local file-system) is deleted, effectively turning the upload into a "file move" operation from the local to the remote side.

VersionedDownload: if **true**, and if the other options require the destination local file to be overwritten by the one that's being downloaded, this option instructs AFT! to automatically keep the old version of the same file in a ".ver" sub-directory of the current local directory. This property defaults to **false**.

VersionedUpload: if **true**, and if the other options require the destination remote file to be overwritten by the one that's being uploaded, this option instructs AFT! to automatically keep the old version of the same file in a ".ver" sub-directory of the current remote directory. This property defaults to **false**.

VersionsToKeepLocal: used in conjunction with VersionedDownload, this property indicates the number of older versions of each file that AFT! will keep in the local ".ver" sub-directory. Defaults to 3.

VersionsToKeepRemote: used in conjunction with VersionedUpload, this property indicates the number of older versions of each file that AFT! will keep in the remote".ver" sub-directory. Defaults to 3.

OTFE: short for On-The-Fly-Encryption, this property indicated whether AFT! should encrypt files as they are uploaded to a remote file server, and decrypt them as they are downloaded back to the local storage. Defaults to **false**.

OTFEKey: a secret word, password or keyword, from which the encryption key will be derived, when the OTFE property is set to **true**. Please do not type your passwords or encryption keys in clear! Use GetSecret instead, or - even better - use the OTFEKeyFromSecret property here below.

OTFEKeyFromSecret: the name of a stored secret (read more about secrets) to be used as seed to derive the encryption key to be used by the On-The-Fly-Encryption algorithm when the OTFE property is set to **true**. If both the OTFEKeyFromSecret and OTFEKey properties are specified, the OTFEKeyFromSecret property prevails.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 scli.Options.UploadPolicy = NeverOverwrite;
 scli.Options.AdjustTimeOnUpload = false;
 scli.Options.StopOnTransferError = true;
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

Directory item

```
DirListItem = {
 Name      // string, fully qualified path and file name (ex:
"/docs/resume.docx")
 Type      // string "FILE" or "DIR"
 Size      // number (64-bit integer)
 TimeStamp // timestamp of the item, in JavaScript "Date()" compatible format
}
```

Every call to a `ListDir` method of any AFT client object will produce an array as a result; each element of such array is an object of `DirListItem` type.
Once a resulting array has been obtained, you can use the typical JavaScript ways to iterate over it, and check the various property of each one of its items.

Example 1 (one way to iterate over a directory list, using a ***`for`*** *cycle):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDir('/docs');
  for (var i = 0; i < dirList.length; i++) {
   Log(dirList[i].Name);
  }
  scli.Close();
 }
 scli = null
}
```

Example 2 (a different way to iterate over a directory list, using `forEach`*):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDir('/docs');
  dirList.forEach(myFunction);
  function myFunction(item, index, array) {
   Log(item.Name + ' [' + item.Size + ' bytes] [' + item.Type + ']');
  }
  scli.Close();
 }
 scli = null
}
```

Sorting a directory list (for all client objects)

```
function SortDir(dirList, sortBy, direction);
```

This function sorts (re-orders) a directory list previously retrieved by either one of the following client object methods (regardless of which client object has generated the list):`ListDir`, `ListDirR`, `ListFiles`,

`ListFilesR`.

Here's an explanation of the parameters this function takes:

- `dirList` is a JavaScript array in which each item is of DirListItem type, typically this list is the result of a call to a client method that retrieves a directory/file list, as explained here above
- `sortBy` is a **`string`** parameter and can be either one of the following:
  - `"name"` sorts the list by file name
  - `"size"` sorts the list by file size
  - `"time"` sorts the list by the timestamp of each file
- `direction` is an **optional** parameter; it can be either `Ascending` or `Descending`. If this parameter is not specified, `Ascending` will be assumed by default.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDir('/docs', All);
  SortDir(dirList, "time", Descending); // sorts from newest to oldest item
  scli.Close();
 }
 scli = null
}
```

AWS S3 client object

`S3Client() // object constructor`

This function creates and returns a new `S3Client` object, which is the AFT object that implements and carries out all file transfers and related operations using Amazon's S3 protocol.

Here's the protocol-specific properties that you may have to initialize/configure before calling the `.Connect()` method:

```
Region                 string
Bucket                 string
APIKeyID               string
APIKeySecret           string
APIKeySecretFromSecret string
UseMetadataWhenListing bool
```

`Region`: this is the AWS region, as specified in AWS' own documentation. For example `'us-east-1'`.

`Bucket`: this is the name of the S3 bucket that you have chosen and assigned to your bucket when you have originally created it.

`APIKeyID`: this is the APIKeyID that you have generated in your AWS account to access this bucket.

`APIKeySecret`: every APIKeyID has a corresponding secret in your AWS account. This is such secret. We do not recommend, though, that you write your secrets in clear in your script code, so either use the `GetSecret` function to populate this value, or use the `APIKeySecretFromSecret` property here below (better choice).

`APIKeySecretFromSecret`: if you have stored your `APIKeySecret` as aSyncplify.me AFT! secret (read more about secrets) then you can set this property to the **name of the secret** in Syncplify.me AFT!'s

database; this is the safest choice.

`UseMetadataWhenListing`: this property is by default set to **`false`** to optimize the speed of your directory listing and file searching commands. Setting this property to**`true`** will slow down directory lists and file searches by a remarkable factor, but it will - in turn - enable the acquisition of the object's metadata, which include the correct TimeStamp of the last modification date of the object. Unfortunately S3 doesn't return such information in the normal directory list, so this option necessarily needs to be set to **`true`** if your script is using the `AdjustTimeOnUpload` or `AdjustTimeOnDownload` options. This is the only way to make sure that the returned TimeStamp matches the custom TimeStamp you set, and not the timestamp that AWS arbitrarily assigns to the object.

Please, keep in mind that once created you (the programmer) have the responsibility of freeing the memory allocated by the object at the end of its use. You can easily do so by simply setting the object to **`null`** once you're done using it (as shown in the example below).

Example:

```
{
 var scli = new S3Client();
 scli.Region = 'us-east-1';
 scli.Bucket = 'yourbucketname';
 scli.APIKeyID = 'wvb8ye5485ye4y7585';
 scli.APIKeySecretFromSecret = 'your_aws_s3_secret_name';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

Azure Blob Storage client object

```
AzureClient() // object constructor
```

This function creates and returns a new `AzureClient` object, which is the AFT object that implements and carries out all file transfers and related operations using Microsoft Azure Blob Storage protocol.

Here's the protocol-specific properties that you may have to initialize/configure before calling the `.Connect()` method:

```
Container              string
AccountName            string
AccountKey             string
AccountKeyFromSecret   string
UseMetadataWhenListing bool
```

`Container`: this is the name of the Azure Blob container that you have chosen and assigned to your container when you have originally created it (other cloud vendors call this "bucket" but in Azure's terminology it's called "Container".

`AccountName`: this is the Account Name for this Azure Storage Account.

`AccountKey`: every AccountName has a corresponding secret AccountKey in your Azure account. This is such key. We do not recommend, though, that you write your secrets in clear in your script code, so either use the `GetSecret` function to populate this value, or use the `AccountKeyFromSecret` property here below (better choice).

`AccountKeyFromSecret`: if you have stored your `AccountKey` as aSyncplify.me AFT! secret (read

more about secrets) then you can set this property to the **name of the secret** in Syncplify.me AFT!'s database; this is the safest choice.

`UseMetadataWhenListing`: this property is by default set to **`false`** to optimize the speed of your directory listing and file searching commands. Setting this property to **`true`** will slow down directory lists and file searches by a remarkable factor, but it will - in turn - enable the acquisition of the object's metadata, which include the correct TimeStamp of the last modification date of the object. Unfortunately Azure doesn't return such information in the normal directory list, so this option necessarily needs to be set to **`true`** if your script is using the `AdjustTimeOnUpload` or `AdjustTimeOnDownload` options. This is the only way to make sure that the returned TimeStamp matches the custom TimeStamp you set, and not the timestamp that AWS arbitrarily assigns to the object.

Important note: as opposed to other cloud platforms, in Azure Blob Storage all folders are "virtual". They exist only when there's at least one blob in them, and disappear by themselves when they're emptied. This implies the following peculiarities in the `AzureClient` object behavior that differ from all other AFT! client objects:

- `MakeDir` always succeeds (returns **`true`**) but truthfully it does absolutely nothing
- `RenDir` is not supported and always return **`false`** (a non-existent object cannot be renamed)

Please, keep in mind that once created you (the programmer) have the responsibility of freeing the memory allocated by the object at the end of its use. You can easily do so by simply setting the object to **`null`** once you're done using it (as shown in the example below).

Example:

```
{
 var cli = new AzureClient();
 cli.Container = 'yourblobcontainername';
 cli.AccountName = 'your_account_name';
 cli.AccountKeyFromSecret = 'your_secret_name';
 if (cli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  cli.Close();
 }
 cli = null
}
```

Google Cloud Storage client object

```
GCSClient() // object constructor
```

This function creates and returns a new `GCSClient` object, which is the AFT object that implements and carries out all file transfers and related operations using Google Cloud Storage protocol.

Here's the protocol-specific properties that you may have to initialize/configure before calling the `.Connect()` method:

```
Bucket                string
CredentialsFile       string //optional
UseMetadataWhenListing bool
```

`Bucket`: this is the name of the S3 bucket that you have chosen and assigned to your bucket when you have originally created it.

`CredentialsFile`: unlike other cloud vendors, Google Cloud exports OAuth2 credentials as a JSON file (read this: https://cloud.google.com/docs/authentication/getting-started). Once you have this JSON file, you have 2 options:

1. either you specify its full path in this `CredentialsFile` property of the GCSClient object, or

2. you export its path as an environment variable named *GOOGLE_APPLICATION_CREDENTIALS* (if you choose this option #2 you don't need to specify the file's path in your script, in fact Syncplify.me AFT! will detect it automatically from your environment variables

`UseMetadataWhenListing`: this property is by default set to **`false`** to optimize the speed of your directory listing and file searching commands. Setting this property to **`true`** will slow down directory lists and file searches by a remarkable factor, but it will - in turn - enable the acquisition of the object's metadata, which include the correct TimeStamp of the last modification date of the object. Unfortunately Azure doesn't return such information in the normal directory list, so this option necessarily needs to be set to **`true`** if your script is using the `AdjustTimeOnUpload` or `AdjustTimeOnDownload` options. This is the only way to make sure that the returned TimeStamp matches the custom TimeStamp you set, and not the timestamp that AWS arbitrarily assigns to the object.

Please, keep in mind that once created you (the programmer) have the responsibility of freeing the memory allocated by the object at the end of its use. You can easily do so by simply setting the object to **`null`** once you're done using it (as shown in the example below).

Example (with explicit location of the credentials file):

```
{
 var cli = new GCSClient();
 cli.Bucket = 'yourbucketname';
 cli.CredentialsFile = 'C:\\MySecretFolder\\GoogleCloudCreds.json';
 if (cli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  cli.Close();
 }
 cli = null
}
```

Example (that assumes the existence of the GOOGLE_APPLICATION_CREDENTIALS environment valiable):

```
{
 var cli = new GCSClient();
 cli.Bucket = 'yourbucketname';
 if (cli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  cli.Close();
 }
 cli = null
}
```

SFTP client object

`SftpClient() // object constructor`

This function creates and returns a new `SftpClient` object, which is the AFT object that implements and carries out all file transfers and related operations using the SFTP protocol.

Here's the protocol-specific properties that you may have to initialize/configure before calling the `.Connect()` method:

```
Host           string
User           string
Pass           string
PassFromSecret string
KeyFile        string
KeyFilePass    string
KeyFileSecret  string
```

`Host`: is the IP address or HostName and Port of the remote file server. Should always be specified in the form `IP:Port` or `Host:Port` format. Examples: `192.168.2.23:22` or `sftp.mycompany.com:22`.

`User`: this is the username to access the remote server. If the server uses APIKey/APISecret instead of Username/Password, this property will contain the APIKey.

`Pass`: this is the password to access the remote server. If the server uses APIKey/APISecret instead of Username/Password, this property will contain the APISecret.

`PassFromSecret`: Syncplify.me AFT! allows you to store secrets (strings) in its encrypted database, so you don't have to put them in clear in your scripts. If you have stored a host's password (or APISecret) as an encrypted secret in Syncplify.me AFT! you can reference it via `PassFromSecret` to retrieve it at runtime without typing it in plain-text in your script. Read more about secrets.

`KeyFile`: if you specify the fully qualified path to a file containing your private key (in RSA format) then the client object will attempt PKI authentication. Leave this property empty/blank to authenticate via simple username and password.

`KeyFilePass`: if you have specified a `KeyFile`, and if the `KeyFile` is password-protected (encrypted), this is the password that's necessary to decrypt such file.

`KeyFileSecret`: if you have specified a `KeyFile`, and if the `KeyFile` is password-protected (encrypted), this is the name of the secret object in Syncplify.me AFT!'s database that corresponds to the `KeyFile`'s password, so you don't have to type such password in clear in your script. Using "secrets" is always the recommended method to specify passwords in Syncplify.me AFT! Readmore about secrets.

Please, keep in mind that once created you (the programmer) have the responsibility of freeing the memory allocated by the object at the end of its use. You can easily do so by simply setting the object to **`null`** once you're done using it (as shown in the example below).

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

FTP(E/S) client object

```
FtpsClient() // object constructor
```

This function creates and returns a new `FtpsClient` object, which is the AFT object that implements and carries out all file transfers and related operations using the FTP (plain), FTPS (implicit SSL/TLS), and

FTPES (explicit SSL/TLS) protocols.

Here's the protocol-specific properties that you may have to initialize/configure before calling the `.Connect()` method:

```
Host               string
User               string
Pass               string
PassFromSecret     string
TLS                TLSMode // TLSNone || TLSExplicit || TLSImplicit
TrustInsecureCerts boolean
```

`Host`: is the IP address or HostName and Port of the remote file server. Should always be specified in the form `IP:Port` or `Host:Port` format. Examples: `192.168.2.23:21` or `ftp.mycompany.com:21`.

`User`: this is the username to access the remote server. If the server uses APIKey/APISecret instead of Username/Password, this property will contain the APIKey.

`Pass`: this is the password to access the remote server. If the server uses APIKey/APISecret instead of Username/Password, this property will contain the APISecret.

`PassFromSecret`: Syncplify.me AFT! allows you to store secrets (strings) in its encrypted database, so you don't have to put them in clear in your scripts. If you have stored a host's password (or APISecret) as an encrypted secret in Syncplify.me AFT! you can reference it via `PassFromSecret` to retrieve it at runtime without typing it in plain-text in your script. Read more about secrets.

`TLS`: this indicates whether or not you wish this client object to use SSL/TLS network encryption, and how. The possible values are:

- `TLSNone`: indicates that no encryption will be used, data will be transferred in clear (not recommended)
- `TLSExplicit`: this is the most common mode, the client object connects in clear, and immediately switches to TLS using the STARTTLS command (this is also the only TLS mode that's actually a recognized standard)
- `TLSImplicit`: assumes that the FTP server is listening on a SSL/TLS socket, so the connection will be established already encrypted (although this may sound more secure than *TLSExplicit* to the untrained ear, it actually isn't even a recognized standard, but we do support it if you wish to use it)

`TrustInsecureCerts`: defaults to **`false`**, but you may set it to **`true`** if you want the client object to accept TLS certificates from servers that can't be verified or that flat out fail verification; this is useful only when you know for sure that the FTP server is using a self-signed or otherwise invalid certificate but you are absolutely sure you can still trust it

Please, keep in mind that once created you (the programmer) have the responsibility of freeing the memory allocated by the object at the end of its use. You can easily do so by simply setting the object to **`null`** once you're done using it (as shown in the example below).

Example:

```
{
 var cli = new FtpsClient();
 cli.Host = 'your.sftpserver.com:21';
 cli.User = 'someusername';
 cli.PassFromSecret = 'my_secret_name';
 cli.TLS = TLSExplicit;
 if (cli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  cli.Close();
 }
 cli = null
}
```

Foreword

All AFT client objects, regardless of the protocol they implement (SFTP, FTP, S3, GCP, ...) implement the same file-transfer-specific methods, so you don't have to worry about learning tons of different methods and function names, and how to use them. All client objects behave the same, and all of them share the same method interfaces, to provide you with a consistent programming experience across all supported file transfer protocols.

Connect

```
function Connect()
```

The `Connect()` function tries to connect and authenticate to the remote service.
If such connection and authentication are successful, this function returns **`true`**, otherwise it returns **`false`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

Disconnect

```
function Close()
```

The `Close()` function of each AFT client object disconnects from the remote service.
If the disconnection was carried out gracefully, it returns **`true`**, otherwise if some errors occurred during disconnection it returns **`false`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

Directory list

```
function ListDir(directory, include)
```

The `ListDir` method retrieves the list of objects, typically files and other directories, contained inside the specified directory.
The result of this function is returned as a JavaScript `array` of objects. Each object is of `DirListItem` type.
Once a resulting array has been obtained, you can use the typical JavaScript ways to iterate over it, and check the various property of each one of its items.

The `include` parameter is used to determine what the `ListDir` command should return, and can have either one of the following 3 values:

- `All`: all files and sub-directories inside `directory` will be returned
- `FilesOnly`: only files contained inside `directory` will be returned
- `DirsOnly`: only sub-directories inside `directory` will be returned

This function is NOT recursive, so only items that are contained inside `directory` will be returned. Items contained in sub-directories of `directory` will not be included in the results. For a recursive version of this function, please, look at `ListDirR`.

Example 1 (one way to iterate over a directory list, using a ***for*** *cycle):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDir('/docs', All);
  for (var i = 0; i < dirList.length; i++) {
   Log(dirList[i].Name);
  }
  scli.Close();
 }
 scli = null
}
```

Example 2 (a different way to iterate over a directory list, using `forEach`*):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDir('/docs', FilesOnly);
  dirList.forEach(myFunction);
  function myFunction(item, index, array) {
   Log(item.Name + ' [' + item.Size + ' bytes] [' + item.Type + ']');
  }
  scli.Close();
 }
 scli = null
}
```

Directory list (recursive)

```
function ListDirR(directory, include)
```

The `ListDirR` method retrieves the list of objects, typically files and other directories, contained inside the specified directory, and all of its sub-directories.
The result of this function is returned as a JavaScript `array` of objects. Each object is of `DirListItem` type.
Once a resulting array has been obtained, you can use the typical JavaScript ways to iterate over it, and check the various property of each one of its items.

The `include` parameter is used to determine what the `ListDirR` command should return, and can have either one of the following 3 values:

- `All`: all files and sub-directories inside `directory` (and inside its sub-directories) will be returned
- `FilesOnly`: only files contained inside `directory` (and inside its sub-directories)will be returned
- `DirsOnly`: only sub-directories inside `directory` (and inside its sub-directories)will be returned

This function is recursive, therefore it will return all matching items from `directory` as well as from all of `directory`'s sub-directories.

Example 1 (one way to iterate over a directory list, using a ***for*** *cycle):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDirR('/docs', All);
  for (var i = 0; i < dirList.length; i++) {
   Log(dirList[i].Name);
  }
  scli.Close();
 }
 scli = null
}
```

Example 2 (a different way to iterate over a directory list, using `forEach`*):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListDirR('/docs', FilesOnly);
  dirList.forEach(myFunction);
  function myFunction(item, index, array) {
   Log(item.Name + ' [' + item.Size + ' bytes] [' + item.Type + ']');
  }
  scli.Close();
 }
 scli = null
}
```

Search for files

```
function ListFiles(directory, mask)
```

The `ListFiles` method retrieves the list of files (not sub-directories) contained inside the specified directory.
The result of this function is returned as a JavaScript `array` of objects. Each object is of `DirListItem` type.
Once a resulting array has been obtained, you can use the typical JavaScript ways to iterate over it, and check the various property of each one of its items.

The `directory` parameter must specify an existing directory.
The `mask` parameter must specify to file-mask to match while searching, wildcards are supported (for example: `*.docx`).

This function is NOT recursive, so only files that are contained inside `directory` will be returned. Files contained in sub-directories of `directory` will not be included in the results. For a recursive version of this function, please, look at `ListFilesR`.

Example 1 (one way to iterate over a directory list, using a ***for*** *cycle):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListFiles('/docs', '*.docx');
  for (var i = 0; i < dirList.length; i++) {
   Log(dirList[i].Name);
  }
  scli.Close();
 }
 scli = null
}
```

Example 2 (a different way to iterate over a directory list, using `forEach`*):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListFiles('/docs', '*.xlsx');
  dirList.forEach(myFunction);
  function myFunction(item, index, array) {
   Log(item.Name + ' [' + item.Size + ' bytes] [' + item.Type + ']');
  }
  scli.Close();
 }
 scli = null
}
```

Search for files (recursive)

```
function ListFilesR(directory, mask)
```

The `ListFilesR` method retrieves the list of files (not sub-directories) contained inside the specified directory and all of its sub-directories.
The result of this function is returned as a JavaScript `array` of objects. Each object is of `DirListItem` type.
Once a resulting array has been obtained, you can use the typical JavaScript ways to iterate over it, and check the various property of each one of its items.

The `directory` parameter must specify an existing directory.
The `mask` parameter must specify to file-mask to match while searching, wildcards are supported (for example: `*.docx`).

This function is recursive, therefore it will return all matching files from `directory` as well as from all of `directory`'s sub-directories.

Example 1 (one way to iterate over a directory list, using a ***for*** *cycle):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListFilesR('/docs', '*.docx');
  for (var i = 0; i < dirList.length; i++) {
   Log(dirList[i].Name);
  }
  scli.Close();
 }
 scli = null
}
```

Example 2 (a different way to iterate over a directory list, using `forEach`*):*

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  dirList = scli.ListFilesR('/docs', '*.xlsx');
  dirList.forEach(myFunction);
  function myFunction(item, index, array) {
   Log(item.Name + ' [' + item.Size + ' bytes] [' + item.Type + ']');
  }
  scli.Close();
 }
 scli = null
}
```

File/directory existence and metadata

```
function Stat(remoteObj)
```

Every client object has a method called Stat which tries to retrieve information about a remote object (file or directory) and, in doing so, checks whether such object exists or not.

This method always returns an object with the following structure:

```
{
  Valid  bool
  Exists bool
  Stat   DirListItem
}
```

Here's an example of a fully populated object of the above type.

```
{
  Valid  : true,
  Exists : true,
  Stat   : {
```

```
    Name: "/docs/budget.xlsx",
    Type: "FILE",
    Size: 526546,
    TimeStamp: 1576095950
  }
}
```

`Valid` is **`true`** when the `.Stat` command has returned valid information. Does not necessarily mean that the object exists, only that the command was executed without issues.

`Exists` is **`true`** if `remoteObj` (a file or directory on the remote file server) exists.

If and only if `Exists` is **`true`**, then the object's `Stat` property will contain meaningful information about the analyzed file/directory.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  var res = scli.Stat('');
  if ((res.Valid) && (res.Exists)) {
   Log('The file exists and its size is '+res.Stat.Size);
  }
  // ...
  scli.Close();
 }
 scli = null
}
```

Make a directory

`function` `MakeDir(directory)`

The `MakeDir` function creates a directory. It also creates, if needed, all directories in the path into which the *leaf* directory is in. Suppose, for example, that a "docs" directory exists in the root of the remote file server, and you `MakeDir('/docs/personal/spreadsheets')`, the MakeDir function will proceed to create the "personal" directory inside the "docs" directory, and then the "spreadsheets" directory inside the "personal" directory, all with a single function call.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.MakeDir('/docs/personal/spreadsheets');
  // ...
  scli.Close();
 }
 scli = null
}
```

Rename a directory

```
function RenDir(currentDirName, newDirName)
```

The `RenDir` function renames a directory. The `currentDirName` directory must exist, and the `newDirName` directory must not exist, otherwise this command will fail and return **`false`**. If the command succeeds, and the directory is successfully renamed, then this function will return **`true`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.RenDir('/docs/yours', '/docs/mine');
  // ...
  scli.Close();
 }
 scli = null
}
```

Delete a directory

```
function DelDir(directory)
```

The `DelDir` function deletes a directory, if it's empty.
This function returns **`true`** if the directory was successfully deleted. If the directory isn't empty, or if for any other reason it could not be deleted, this function returns **`false`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.DelDir('/docs/some_empty_directory');
  // ...
  scli.Close();
 }
 scli = null
}
```

Delete a directory tree

```
function DelTree(directory)
```

The `DelTree` function deletes a directory tree, by recursively deleting also all files and sub-directories in it.
This function returns **`true`** if the directory tree was successfully deleted. If for any other reason it could not be deleted, this function returns **`false`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.DelTree('/docs/some_directory');
  // ...
  scli.Close();
 }
 scli = null
}
```

Delete a file

```
function Delete(what)
```

The `Delete` method attempts to delete a file from the remote server, and returns **`true`** if deletion was successful or **`false`** if it wasn't.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.Delete('/docs/some_document.docx');
  // ...
  scli.Close();
 }
 scli = null
}
```

Rename/move an object

```
function Rename(what, toWhere)
```

The `Rename` function renames or moves a file system object (file or folder) on the remote file server. It returns **`true`** if the object is successfully renamed/moved, otherwise it returns**`false`**.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.Rename('/docs/bio.docx', '/docs/bio.docx.bak');
  // ...
  scli.Close();
 }
 scli = null
}
```

Upload files

```
function Upload(what, toWhere)
```

The `Upload` function upload the file(s) specified in the `what` parameter (supports wildcards) to the `toWhere` destination directory on the remote file server. This function is NOT recursive, therefore when using wildcards, this function will NOT upload matching files from sub-directories of the directory specified in the `what` parameter. For a recursive version of this function see the`UploadR` function.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.Upload('./docs/*.docx', '/archive/docs');
  // ...
  scli.Close();
 }
 scli = null
}
```

Upload file with path

```
function UploadWithPath(what, toWhere, skip)
```

The `UploadWithPath` function upload the file(s) specified in the `what` parameter (supports wildcards) to the `toWhere` destination directory on the remote file server, retaining the path of the original file and recreating it if necessary. The `skip` parameter is an integer number that instructs the function to skip the first N directories of the original path when rebuilding it into the destination path. This function is NOT recursive, therefore when using wildcards, this function will NOT upload matching files from sub-directories of the directory specified in the `what` parameter. For a recursive version of this function see the `UploadWithPathR` function.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.UploadWithPath('./data/docs/*.docx', '/archive', 0); // uploads files
to /archive/data/docs
  // ...
  scli.UploadWithPath('./data/docs/*.docx', '/archive', 1); // skip=1 means
uploads files to /archive/docs
  // ...
  scli.Close();
 }
 scli = null
}
```

Upload files recursively

```
function UploadR(what, toWhere)
```

The `UploadR` function upload the file(s) specified in the `what` parameter (supports wildcards) to the `toWhere` destination directory on the remote file server. This function is recursive, therefore when using wildcards, this function will also upload all matching files from sub-directories of the directory specified in

the `what` parameter.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.UploadR('./docs/*.docx', '/archive/docs');
  // ...
  scli.Close();
 }
 scli = null
}
```

Upload files with path recursively

```
function UploadWithPathR(what, toWhere, skip)
```

The `UploadWithPathR` function upload the file(s) specified in the `what` parameter (supports wildcards) to the `toWhere` destination directory on the remote file server, retaining the path of the original file and recreating it if necessary. The `skip` parameter is an integer number that instructs the function to skip the first N directories of the original path when rebuilding it into the destination path. This function is recursive, therefore when using wildcards, this function will also upload all matching files from sub-directories of the directory specified in the `what` parameter.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.UploadWithPathR('./data/docs/*.docx', '/archive', 0); // uploads files
to /archive/data/docs and subfolders
  // ...
  scli.UploadWithPathR('./data/docs/*.docx', '/archive', 1); // skip=1 means
uploads files to /archive/docs and subfolders
  // ...
  scli.Close();
 }
 scli = null
}
```

Download files

```
function Download(what, toWhere)
```

The `Download` function download the file(s) specified in the `what` parameter (supports wildcards) from the remote file server to the `toWhere` destination directory on the local file system. This function is NOT

recursive, therefore when using wildcards, this function will NOT download matching files from sub-directories of the directory specified in the `what` parameter. For a recursive version of this function see the `DownloadR` function.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.Download('/archive/docs/*.docx', './docs');
  // ...
  scli.Close();
 }
 scli = null
}
```

Download files with path

```
function DownloadWithPath(what, toWhere)
```

The `DownloadWithPath` function download the file(s) specified in the `what` parameter (supports wildcards) from the remote file server to the `toWhere` destination directory on the local file system, retaining the path of the original file and recreating it if necessary. The `skip` parameter is an integer number that instructs the function to skip the first N directories of the original remote path when rebuilding it into the local destination path. This function is NOT recursive, therefore when using wildcards, this function will NOT download matching files from sub-directories of the directory specified in the`what` parameter. For a recursive version of this function see the `DownloadWithPathR` function.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.DownloadWithPath('/arc/docs/*.docx', './here', 0); // downloads files
to ./here/arc/docs
  // ...
  scli.DownloadWithPath('/arc/docs/*.docx', './here', 1); // skip=1 downloads
files to ./here/docs
  // ...
  scli.Close();
 }
 scli = null
}
```

Download files recursively

```
function DownloadR(what, toWhere)
```

The `DownloadR` function download the file(s) specified in the `what` parameter (supports wildcards) from the remote file server to the `toWhere` destination directory on the local file system. This function is

recursive, therefore when using wildcards, this function will also download matching files from sub-directories of the directory specified in the `what` parameter.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.DownloadR('/archive/docs/*.docx', './docs');
  // ...
  scli.Close();
 }
 scli = null
}
```

Download files with path recursively

```
function DownloadWithPathR(what, toWhere)
```

The `DownloadWithPathR` function download the file(s) specified in the `what` parameter (supports wildcards) from the remote file server to the `toWhere` destination directory on the local file system, retaining the path of the original file and recreating it if necessary. The `skip` parameter is an integer number that instructs the function to skip the first N directories of the original remote path when rebuilding it into the local destination path. This function is recursive, therefore when using wildcards, this function will also download matching files from sub-directories of the directory specified in the`what` parameter.

Example:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.KeyFile = './my_id.rsa';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  scli.DownloadWithPathR('/arc/docs/*.docx', './here', 0); // downloads files
to ./here/arc/docs and subfolders
  // ...
  scli.DownloadWithPathR('/arc/docs/*.docx', './here', 1); // skip=1 downloads
files to ./here/docs and subfolders
  // ...
  scli.Close();
 }
 scli = null
}
```

Create a remote file system watcher

```
RemoteWatcher(clientObject) // object constructor
```

This function creates and returns a new Remote Watcher object using the specified client object remote connection. This object can be used later in the script to be constantly notified of the desired changes to the observed (watched) directory/folder (and optionally its sub-folders) on the remote file server.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.NotifyRemove = false;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Watch a directory for changes

```
function WatchDir(path, recursive)
```

This RemoteWatcher method adds a directory/folder to the list of directories/folders "watched" by the file system watcher.
The `path` parameter is a string, and specifies a directory to be watched.
The recursive parameter is a boolean, and specifies whether or not the RemoteWatcher should also watch all sub-folders of the specified folder.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.NotifyRemove = false;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
```

```
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Choose events to watch

```
.NotifyCreate // boolean - default: true - triggered when an object is created
.NotifyRemove // boolean - default: true - triggered when an object is removed
.NotifyModify // boolean - default: true - triggered when an object is
modified
```

These 3 properties of the RemoteWatcher object determine which remote file-system events will be included in the watcher's notifications and which ones won't. By default they are all set to true, so unless you want to disable some of them, you don't need to set/reset them in your code.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.NotifyRemove = false;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Delay notifications

```
.DelayBySeconds // integer number
```

This property of the RemoteWatcher object causes a delay of the specified number of seconds to all file-system event notifications. This can be extremely useful to allow the OS enough time to complete file operations before the RemoteWatcher notifies our script and triggers the execution of a file transfer operation, for example.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.DelayBySeconds = 25;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Inclusion/exclusion filters

```
.InclusionFilter // array of strings
.ExclusionFilter // array of strings
```

These two properties instruct the RemoteWatcher to include or exclude specific file-masks.
By default, `InclusionFilter` is `['*']`, so it includes everything, and `ExclusionFilter` is empty, so nothing will be excluded.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
```

```
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.DelayBySeconds = 25;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Start the remote watcher

```
function Start()
```

This RemoteWatcher method triggers the asynchronous execution of the underlying remote file-system watcher, so that your script can subsequently poll it for pending events. No notification will be available to the script prior to calling this method.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher(scli);
  watchr.WatchDir('/Docs', true);
  watchr.DelayBySeconds = 25;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
```

```
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Poll the remote watcher event queue

```
function Events() // returns an array of event objects
```

Each event object is defined as follows:

```
TimeStamp JSTime // timestamp of when the event happened (in JavaScript Date() compatible format)
Event     string // file system event as string: CREATE, REMOVE, MODIFY
Object    string // the remote object (file or directory) affected by Event
```

This FsWatcher method is designed to be called within the scope of an endless loop, to keep the script running forever (unless terminated by an admin or an OS signal). Each time this method is called, it returns an array of pending, to-be-handled, file system notifications. Every call to this method will also clear the pending notifications, so all events that are left un-handled will not be notified again.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient(scli);
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new RemoteWatcher();
  watchr.WatchDir('/Docs', true);
  watchr.DelayBySeconds = 25;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(30000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'CREATE') {
      scli.DownloadWithPath(evt[i].Object, 'C:\\MyLocalCopies', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Create a local file system watcher

```
FsWatcher() // object constructor
```

This function creates and returns a new File-System Watcher object. This object can be used later in the script to be constantly notified of the desired changes to the observed (watched) directory/folder, and

optionally its sub-folders.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.NotifyRename = false;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Watch a directory for changes

```
function WatchDir(path, recursive)
```

This FsWatcher method adds a directory/folder to the list of directories/folders "watched" by the file system watcher.
The `path` parameter is a string, and specifies a directory to be watched.
The recursive parameter is a boolean, and specifies whether or not the FsWatcher should also watch all sub-folders of the specified folder.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.NotifyRename = false;
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
```

```
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Choose events to watch

```
.NotifyCreate // boolean - default: true - triggered when an object is created
.NotifyWrite  // boolean - default: true - triggered when an object is written
.NotifyRemove // boolean - default: true - triggered when an object is removed
.NotifyRename // boolean - default: true - triggered when an object is renamed
.NotifyChmod  // boolean - default: true - triggered when an object's metadata
is changed
```

These 5 properties of the FsWatcher object determine which file system events will be included in the watcher's notifications and which ones won't. By default they are all set to true, so unless you want to disable some of them, you don't need to set/reset them in your code.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.NotifyRename = false; // will NOT notify File-Rename events
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
```

```
  scli.Close();
 }
 scli = null;
}
```

Delay notifications

```
.DelayBySeconds // integer number
```

This property of the FsWatcher object causes a delay of the specified number of seconds to all file-system event notifications. This can be extremely useful to allow the OS enough time to complete file operations before the FsWatcher notifies our script and triggers the execution of a file transfer operation, for example.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.DelayBySeconds = 10 // delay all notifications by 10 seconds
  watchr.InclusionFilter = ['*.docx', '*.xlsx']
  watchr.ExclusionFilter = ['some_private_document.docx']
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Inclusion/exclusion filters

```
.InclusionFilter // array of strings
.ExclusionFilter // array of strings
```

These two properties instruct the FsWatcher to include or exclude specific file-masks.
By default, `InclusionFilter` is `['*']`, so it includes everything, and `ExclusionFilter` is empty, so nothing will be excluded.

Example:

```
{
 ConsoleFeedback = true;
```

```
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.InclusionFilter = ['*.docx', '*.xlsx'] // include only *.docx and
*.xlsx files
  watchr.ExclusionFilter = ['some_private_document.docx'] // exclude this one
specific file
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Start watching for events

```
function Start()
```

This FsWatcher method triggers the asynchronous execution of the underlying file-system watcher, so that your script can subsequently poll it for pending events. No notification will be available to the script prior to calling this method.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.Start(); // from this point on, notification will be available
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events()
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
```

```
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Poll the file system event queue

```
function Events() // returns an array of event objects
```

Each event object is defined as follows:

```
TimeStamp JSTime // timestamp of when the event happened (in JavaScript Date() compatible format)
Event     string // file system event as string: CREATE, WRITE, REMOVE, RENAME, CHMOD
Object    string // the file system object (file or directory) affected by Event
```

This FsWatcher method is designed to be called within the scope of an endless loop, to keep the script running forever (unless terminated by an admin or an OS signal). Each time this method is called, it returns an array of pending, to-be-handled, file system notifications. Every call to this method will also clear the pending notifications, so all events that are left un-handled will not be notified again.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = new FsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break;
   }
   evt = watchr.Events() // gets the list of pending events to be handled
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

List a local directory

```
function ListDir(what, mask)
```

`ListDir` lists the contents of a directory in the local file system.

This functions accepts either 1 or 2 parameters:

- `what` must be a valid and existent directory on a local file system, this parameter is mandatory
- `mask` is a file-mask (example: `"*.docx"`) to limit the scope of the returned results only to items matching such mask

This function is NOT recursive, therefore it only returns matching items from the specific directory identified by the `what` parameter, but NOT items contained in sub-directories of the `what` directory.

List a local directory (recursive)

```
function ListDirR(what, mask)
```

`ListDirR` lists the contents of a directory, and all of its sub-directories, in the local file system.

This functions accepts either 1 or 2 parameters:

- `what` must be a valid and existent directory on a local file system, this parameter is mandatory
- `mask` is a file-mask (example: `"*.docx"`) to limit the scope of the returned results only to items matching such mask

This function is recursive, therefore it returns all matching items from the specific directory identified by the `what` parameter, as well as items contained in sub-directories of the `what` directory.

Copy a local file

```
function CopyFile(what, toWhere)
```

This function copies the `what` file to the `toWhere` destination directory in the local file system.

Move a local file

```
function MoveFile(what, toWhere)
```

This function moves the `what` file to the `toWhere` destination directory in the local file system.

Delete a file

```
function DelFile(what)
```

This function deletes the `what` file from the local file system.

Securely erase a file

```
function SecureErase(fileName, numPasses)
```

This function deletes the `fileName` file from the local file system using a secure erasure algorithm. This means that the file will be overwritten with crypto-secure pseudo-random data before it's actually deleted from the storage medium. For this reason, depending on the size of the file, this function may take a while to complete.

The second parameter (`numPasses`) is **optional** and specifies how many times the file needs to be overwritten with crypto-secure pseudo-random data before the actual deletion occurs.

As almost any other function in the aftJS, this function returns a boolean value, `true` if it succeeds, `false` if it fails.

Example (on Windows, without optional parameter):

```
{
  SecureErase('C:\\Data\\SomeFile.docx');
}
```

Example (on Linux, with optional parameter to specify that we want the file to be overwritten 3 times):

```
{
  SecureErase('/home/docs/SomeFile.pdf', 3);
}
```

Create a directory

```
function MakeDir(what)
```

This function attempts to create the `what` directory path on the local (or UNC) file system. If the directory is successfully created or if it already exists this function will return **`true`**, instead if the directory cannot be created this function will fail and return **`false`**.

Delete a directory

```
function DelDir(what)
```

This function attempts to deletes the `what` directory from the local file system. If the directory is empty the function will succeed and return **`true`**, if the directory isn't empty this function will fail and return **`false`**.

Delete a directory tree

```
function DelTree(what)
```

This function attempts to deletes the `what` directory from the local file system. Even if the directory isn't empty, the function will try to recursively delete all files and sub-directories within it. If it succeeds it returns **`true`**, if it fails it returns **`false`**.

Read a text file

```
function ReadTextFile(filename)
```

This function reads a text file and returns its entire contents as a `string`.
If the file doesn't exist or if the operating system returns an I/O error, this function returns an empty string.

Example:

```
{
 var fileContents = ReadTextFile('./docs/somefile.txt');
 Log(fileContents);
}
```

Write some text to file

Syncplify.me AFT! actually does offer 2 distinct functions to write text to file:

```
function WriteTextToFile(filename, text)
function AppendTextToFile(filename, text)
```

Both of this functions write text to a file and return `true` if the operation was successful, otherwise they return `false`.
Also, both functions create the file if it doesn't exist.

The main difference is that the `AppendTextToFile` function will append text at the end of a file (if it exists) while the `WriteTextToFile` will overwrite whatever contents are already in a file with the specified text, and all pre-existing file content will be lost.

Both functions support "escaped strings", so, for example, if you want to write a sentence and then a NEWLINE special control character, you can simply add `\n` to the text to be written to file. String escaping follows this convention.

Example:

```
{
 AppendTextFile('./docs/somefile.txt', 'Hello world!\n');
}
```

Create a zip archive

```
function Zip(what, zipArchive)
```

The `Zip` function creates a compressed archive with the files that are passed to it in the`what` argument. Supports wildcards. If the destination zip archive already exists it will be overwritten and replaced.

Example:

```
{
 Zip('./documents/*.docx', './archives/dox.zip');
}
```

Identify a file MIME-type

```
function FileType(filename)
```

Most file types can be identified regardless of the file name or extension, by simply reading the first 261 bytes of the file itself (no need to read the whole file, so this process is extremely quick and doesn't waste any RAM).

Here's a list of file types that this function can identify:

Image

jpg - image/jpeg
png - image/png
gif - image/gif
webp - image/webp
cr2 - image/x-canon-cr2
tif - image/tiff
bmp - image/bmp
heif - image/heif
jxr - image/vnd.ms-photo
psd - image/vnd.adobe.photoshop

ico - image/x-icon
dwg - image/vnd.dwg

Video

mp4 - video/mp4
m4v - video/x-m4v
mkv - video/x-matroska
webm - video/webm
mov - video/quicktime
avi - video/x-msvideo
wmv - video/x-ms-wmv
mpg - video/mpeg
flv - video/x-flv
3gp - video/3gpp

Audio

mid - audio/midi
mp3 - audio/mpeg
m4a - audio/m4a
ogg - audio/ogg
flac - audio/x-flac
wav - audio/x-wav
amr - audio/amr
aac - audio/aac

Archive

epub - application/epub+zip
zip - application/zip
tar - application/x-tar
rar - application/x-rar-compressed
gz - application/gzip
bz2 - application/x-bzip2
7z - application/x-7z-compressed
xz - application/x-xz
pdf - application/pdf
exe - application/x-msdownload
swf - application/x-shockwave-flash
rtf - application/rtf
iso - application/x-iso9660-image
eot - application/octet-stream
ps - application/postscript
sqlite - application/x-sqlite3
nes - application/x-nintendo-nes-rom
crx - application/x-google-chrome-extension
cab - application/vnd.ms-cab-compressed
deb - application/x-deb
ar - application/x-unix-archive
Z - application/x-compress
lz - application/x-lzip

rpm - application/x-rpm
elf - application/x-executable
dcm - application/dicom

Documents

doc - application/msword
docx - application/vnd.openxmlformats-officedocument.wordprocessingml.document
xls - application/vnd.ms-excel
xlsx - application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
ppt - application/vnd.ms-powerpoint
pptx - application/vnd.openxmlformats-officedocument.presentationml.presentation

Font

woff - application/font-woff
woff2 - application/font-woff
ttf - application/font-sfnt
otf - application/font-sfnt

Application

wasm - application/wasm

Introduction to the HttpCli object

When JavaScript is run inside of the browser, you can use the (non-ECMA) *fetch* function to perform http/https operations. The problem with *fetch*, though, is that it's designed to run inside of an environment (the web browser) that's totally asynchronous by definition. You start fetching something, then the browser goes on to do something else, and when (if ever) the fetched content becomes available, the execution cycle of the *fetch* operation is resumed. This works well in a browser, but would never work inside of a scripting environment where certainty is an absolute requirement when it comes to execution flow.

We have, therefore, added to AFT! our own native http/https client object, called **HttpCli**.
It runs synchronized with the execution environment, ensuring that all functions that rely on the availability of the results of a web request will be correctly serialized. And, for convenience and ease of use, it is possible to configure it using the "fluent paradigm".

Here's the same example three times, the first version is more traditional, the second and third versions use the "fluent paradigm":

First version (non-fluent):

```
{
  var hc = new HttpCli();
  hc.Url("https://www.example.com");
  hc.Timeout(30);
  hc.Header("Custom-Header", "My custom header content");
  var res = hc.Get();
}
```

Second version (fluent):

```
{
  var hc = new HttpCli();
  var res = hc.Url("https://www.example.com").Timeout(30).Header("Custom-
Header", "My custom header content").Get();
}
```

Third version (fluent and folded):

```
{
  var hc = new HttpCli();
  var res = hc.Url("https://www.example.com").
               Timeout(30).
               Header("Custom-Header", "My custom header content").
               Get();
}
```

Learn mode about HttpCli's configuration methods, then learn about its HTTP(S) verb methods and response object.

HttpCli configuration methods

Before you call any of the http/https verbs to actually make the HttpCli do something, you must first configure it according to what you actually intend it to do. This means, for example, specifying the URL you want to call, or the request body you want to send to such URL, or even a timeout past which you want this call to give up.

The HttpCli object is quite flexible in these regards; let's assume your HttpCli client object name is `hc` (you just created it with `var hc = new HttpCli();`)... here's what you can/need-to configure:

`hc.Url(fullyQualifiedUrl) // mandatory; fullyQualifiedUrl is a string`
This configuration is mandatory, every time you want to perform an http/https call, you have to set the URL property, which is the full address of the web resource you're addressing your call to (ex: `hc.Url("https://www.example.com");`).
Calling with method more than once will substitute the previous URL, so only the most recent call to `.Url()` will be considered.

`hc.Accept(contentType) // optional; contentType is a string`
You may set this configuration if you want your HttpCli to only accept from the server a response that has a certain MIME type.
Calling with method more than once will substitute the previous value of Accept, so only the most recent call to `.Accpet()` will be considered.

`hc.ApiKey(yourApiKey) // optional; yourApiKey is a string`
If the server requires an API Key to serve a certain resource, you may specify such API Key using this method.
Calling with method more than once will substitute the previous value of ApiKey, so only the most recent call to `.ApiKey()` will be considered.

`hc.BasicAuth(username, password) // optional; username and passwords are strings`
If the server requires basic authentication to serve a certain resource, you may specify username and password using this method.
Calling with method more than once will substitute the previous value of BasicAuth, so only the most recent call to `.BasicAuth()` will be considered.

`hc.Bearer(bearerToken) // optional; bearerToken is a string`
If the server requires a Bearer Token to serve a certain resource, you may specify such Bearer Token using this method.
Calling with method more than once will substitute the previous value of the Bearer Token, so only the most recent call to `.Bearer()` will be considered.

`hc.FormField(fieldName, fieldValue) // optional; fieldName and fieldValue are strings`
If you want to send your request body with a multipart/form in it (typical with POST request), you may call this method to add a form field and its value to the request body payload. This call is "additive", so you can call `.FormField()` multiple times to add multiple form fields and values.

`hc.Header(headerName, headerValue) // optional; headerName and headerValue are strings`
If you want to send your http/https request with additional headers in it, you may call this method to add a

header and its value to the request itself before it is sent to the server. This call is "additive", so you can call `.Header()` multiple times to add multiple headers to the outgoing http/https call.

```
hc.InsecureSkipVerify() // no parameters
```
Adding InsecureSkipVerify() to your fluent code line instructs the client to accept any server certificate, even self-signed ones, when performing https:// requests.

```
hc.ReqBody(body) // optional; body is a string (or a stringified JSON object)
```
This mathod allows you to specify the raw body payload to be sent with this request; if the string you pass to it is recognized as a valid JSON structure, the HttpCli object will also automatically add/set the "Content-Type" header to "application/json".
Calling with method more than once will substitute the previous body payload, so only the most recent call to `.ReqBody()` will be considered.

```
hc.Timeout(seconds) // optional; seconds is a positive integer
```
This simply sets a timeout past which HttpCli will give up if it hasn't received a response from the server yet.
Calling with method more than once will substitute the previous timeout value, so only the most recent call to `.Timeout()` will be considered.

```
hc.UserAgent(softwareId) // optional; softwareId is a string
```
This method allows you to set a custom User-Agent for your http/https call.
Calling with method more than once will substitute the previous user agent value, so only the most recent call to `.UserAgent()` will be considered.

HttpCli http/https verbs

The HTTP(S) protocol defines the following "verbs" (that typically all web servers honor, although restrictions may apply because of security configurations):

- GET
- POST
- PUT
- PATCH
- DELETE
- HEAD

AFT!'s HttpCli object, therefore has a method for each one of the above verbs, plus one extra method to allow you to send custom verbs to web servers that may be custom-built to support them:

- `.Get()`
- `.Post()`
- `.Put()`
- `.Patch()`
- `.Delete()`
- `.Head()`
- `.Do(customverb) // customverb must be a string`

All methods here above return an HttpRes (http response) object.

Here's a few examples of valid usages of HttpCli's verb methods:

Example #1 (a simple get):
```
{
  var hc = new HttpCli();
  var res = hc.Url("https://www.example.com").Timeout(30).Get();
  if (res.IsValid() && (res.StatusCode() == 200)) {
    Log(res.BodyAsString());
  }
```

```
}
```

Example #2 (post some JSON data):

```
{
  var hc = new HttpCli();
  var res =
hc
.Url
("https://www.some.host").Timeout
(30).ReqBody('{"name":"John","age":42}').Post();
  if (res.IsValid() && (res.StatusCode() == 201)) {
    Log('Success!');
  }
}
```

Example #3 (a custom verb):

```
{
  var hc = new HttpCli();
  // Let's pretend your custom web server supports a "HELLO" verb
  var res = hc.Url("https://www.your.host").Timeout(30).Do("HELLO");
  if (res.IsValid() && (res.StatusCode() == 200)) {
    Log(res.BodyAsString());
  }
}
```

Learn more about how to use HttpCli's response object.

HttpCli response object

Every time an HttpCli verb method is called, it will produce an **HttpRes** object as a result.

Let's consider the following simple script for example:

```
{
  var hc = new HttpCli();
  var res = hc.Url("https://www.example.com").Timeout(30).Get();
  // "res" here above is an HttpRes object with its own methods
}
```

Now, from the example above, the `res` object will have the following methods:

`res.IsValid() // boolean`
This method simply returns **`true`** if the http/https call was completed, or **`false`** otherwise (for example if the call times out this method returns **`false`**)

`res.StatusCode() // integer number`
This method returns the status code resulting from the http/https call, for example if everything went well a `.Get()` request will probably return `200`, while a `.Post()` request will return `201`. Other common and well-known codes are `403` (unauthorized), `404` (not found), and `500` (internal server error). Learn more about HTTP status codes.

`res.BodyAsString() // string`
This method returns the body of the response as a string (useful when the body is a web page or a JSON object for example).

`res.BodyAsBytes() // array of bytes`
This method returns the body of the response as an array of bytes (useful when the body is a binary object, like an image for example).

`res.BodySaveToFile(filepath) // filepath must be a string`
This method saves the body of the response to a file which fully-qualified path is passed as an argument (ex: */downloads/budget.csv*).

`res.ContentType() // string`

This method returns a string containing the MIME Content-Type as reported by the server.

```
res.ContentLength() // integer
```
This method returns the Content-Length as reported by the server (some servers and CDNs fail to report this).

```
res.Encoding() // string
```
This method returns the Content-Transfer-Encoding as reported by the server (this also may be missing in some cases).

```
res.Headers() // JSON object, each property of the object is 1 response header
```
This method returns a single JSON object in which each property represents one response header. Example:


```
{
  "Cache-Control":"max-age=604800",
  "Content-Type":"text/html; charset=UTF-8",
  "Date":"Sun, 30 Aug 2020 16:24:06 GMT",
  "X-Cache":"HIT",
  "Age":"512004",
  "Etag":"\"3147526947+gzip\"",
  "Expires":"Sun, 06 Sep 2020 16:24:06 GMT",
  "Server":"ECS (sjc/4E76)",
  "Last-Modified":"Thu, 17 Oct 2019 07:18:26 GMT",
  "Vary":"Accept-Encoding"
}
```


```
res.Cookies() // array of string/string JSON objects, each object is a cookie
```
This method returns an array of JSON object, in which each object represents a cookie returned by the web server in this response. For example, Google returns an array of cookies like this one:


```
[
  {
    "Name":"1P_JAR",
    "Value":"2020-08-30-16",
    "Path":"/",
    "Domain":".google.com",
    "Expires":"2020-09-29T16:32:11Z",
    "RawExpires":"Tue, 29-Sep-2020 16:32:11 GMT",
    "MaxAge":0,
    "Secure":true,
    "HttpOnly":false,
    "SameSite":0,
    "Raw":"1P_JAR=2020-08-30-16; expires=Tue, 29-Sep-2020 16:32:11 GMT; path=/
; domain=.google.com; Secure",
    "Unparsed":[]
  },
  {
    "Name":"NID",
    "Value":"204=EHXBqKVUuskC5fcv3UnbLPB7oN0LU-
nTODKDhuvBF5WzonZJToiwoBK12N7gr3Pycq8jCZDNS6PW9SV57GtIeCYw488",
    "Path":"/",
    "Domain":".google.com",
    "Expires":"2021-03-01T16:32:11Z",
    "RawExpires":"Mon, 01-Mar-2021 16:32:11 GMT",
    "MaxAge":0,
    "Secure":false,
    "HttpOnly":true,
    "SameSite":0,
    "Raw":"NID=204=EHXBqKVUuskC5fcv3UnbLPB7oN0LU-
nTODKDhuvBF5WzonZJToiwoBK12N7gr3Pycq8jCZDNS6PW9SV57GtIeCYw488; expires=Mon,
01-Mar-2021 16:32:11 GMT; path=/; domain=.google.com; HttpOnly",
    "Unparsed":[]
  }
]
```

AMQP version 0.9.1 and 1.0

There are two milestone versions of the AMQP message queuing protocol, and they are **totally incompatible with each other**:

- AMQP v0.9.1: used by RabbitMQ, StormMQ, Apache Qpid, JORAM, and others...
- AMQP v1.0: used by Apache ActiveMQ, Azure Event Hubs, Azure Service Bus, Solace, and others...

Syncplify.me AFT! features dedicated objects to handle each one of the above protocols, so - please - **make sure you determine the exact protocol** your provider uses, and choose the proper object, otherwise you won't be able to connect to your message queue service.

`AmqpClient091()` is the message queue client for AMQP v0.9.1
`AmqpClient10()` is the message queue client for AMQP v1.0

Both client objects supports both the plain-unencrypted **amqp://** and the secure **amqps://** protocols.

Only the names of the object creator functions differ. All methods of both objects are absolutely identical. Here's two examples, so you can see that the only line that differs is the line to create the correct client object, in every other way these 2 scripts are absolutely identical:

*Script that connects to an **AMQP v0.9.1** (ex: RabbitMQ) and monitors the "myqueue" queue:*

```
{
  ConsoleFeedback = true;

  var cli = new AmqpClient091();

  cli.URL = 'amqp://localhost:5672';
  cli.User = 'guest';
  cli.Pass = 'guest';

  if (cli.Connect()) {
    cli.MonitorQueue('myqueue');
    while (true) {
      Sleep(1000);
      if (HaltSignalReceived()) {
        break;
      }
      var msgs = cli.GetMessages();
      if (msgs.length > 0) {
        Log(JSON.stringify(msgs));
      }
    }
    cli.Close();
  }
  cli = null;
}
```

*Script that connects to an **AMQP v1.0** (ex: ActiveMQ) and monitors the "myqueue" queue:*

```
{
  ConsoleFeedback = true;

  var cli = new AmqpClient10();

  cli.URL = 'amqp://localhost:5672';
  cli.User = 'guest';
  cli.Pass = 'guest';

  if (cli.Connect()) {
    cli.MonitorQueue('myqueue');
```

```
    while (true) {
      Sleep(1000);
      if (HaltSignalReceived()) {
        break;
      }
      var msgs = cli.GetMessages();
      if (msgs.length > 0) {
        Log(JSON.stringify(msgs));
      }
    }
    cli.Close();
  }
  cli = null;
}
```

AMQP client object properties

Both the `AmqpClient091` and the `AmqpClient10` objects have the exact same properties:

```
URL           string // ex: amqp://amqp.myhost.com:5672 or
amqps://amqp.myhost.com:5672
User          string // username if session must authenticate
Pass          string // password (but it's better to use PassFromSecret)
PassFromSecret string // name of the secret to retrieve the password at
runtime
```

Connecting to an AMQP message queue

```
function Connect()
```

Before calling the `Connect()` method of the ***AmqpClinentXX*** object of your choice it is necessary to populate the object properties.

Once the properties are correctly populated, calling the `Connect()` methods attempts to connect to the AMQP message queue, and returns a **`boolean`** value:

- **`true`**: the connection was successful (and you can monitor one or more queues)
- **`false`**: the connection was unsuccessful (and you should not attempt to monitor any queue)

Example:

```
{
  ConsoleFeedback = true;

  var cli = new AmqpClient091();

  cli.URL = 'amqp://localhost:5672';
  cli.User = 'guest';
  cli.Pass = 'guest';

  if (cli.Connect()) {
    cli.MonitorQueue('myqueue');
    while (true) {
      Sleep(1000);
      if (HaltSignalReceived()) {
        break;
      }
      var msgs = cli.GetMessages();
      if (msgs.length > 0) {
        Log(JSON.stringify(msgs));
```

```
      }
    }
    cli.Close();
  }
  cli = null;
}
```

Adding a queue to monitor

```
function MonitorQueue(queueName) // queueName is expected to be a string
```

The `MonitorQueue()` function instructs the client object to start monitoring the specified queue for incoming messages.

Example:

```
{
  ConsoleFeedback = true;

  var cli = new AmqpClient091();

  cli.URL = 'amqp://localhost:5672';
  cli.User = 'guest';
  cli.Pass = 'guest';

  if (cli.Connect()) {
    cli.MonitorQueue('myqueue');
    while (true) {
      Sleep(1000);
      if (HaltSignalReceived()) {
        break;
      }
      var msgs = cli.GetMessages();
      if (msgs.length > 0) {
        Log(JSON.stringify(msgs));
      }
    }
    cli.Close();
  }
  cli = null;
}
```

Processing incoming events/messages

```
function GetMessages() // returns an array of messages received from the queue
```

Each returned message (each item of the array) has the following format:

```
{
  receivedAt // JavaScript date
  queue      // string
  message    // string
}
```

It is also recommended, inside the loop in which `GetMessages()` is called iteratively, to also monitor whether a "halt" request has been issued and the script must terminate.

Example:

```
{
  ConsoleFeedback = true;

  var cli = new AmqpClient091();

  cli.URL = 'amqp://localhost:5672';
  cli.User = 'guest';
  cli.Pass = 'guest';
```

```
  if (cli.Connect()) {
    cli.MonitorQueue('myqueue');
    while (true) {
      Sleep(1000);
      if (HaltSignalReceived()) {
        break;
      }
      var msgs = cli.GetMessages();
      if (msgs.length > 0) {
        Log(JSON.stringify(msgs));
      }
    }
    cli.Close();
  }
  cli = null;
}
```

Send to Slack (webhook)

```
SendToSlackWebHook(
      webhookURL, // string (mandatory)
      message,    // string (mandatory)
      sender,     // string (optional)
      icon        // string (optional)
)
```

This function posts a notification to a Slack channel via Slack's "Incoming WebHooks".

The `webhookURL` and `message` parameters are mandatory. You may, if you wish, also specify a `sender` (free-text string), and an `icon` name using the standard emoji icon name format.

This function returns **`true`** if the notification was successfully posted to the desired Slack channel, otherwise it returns **`false`**.

Example:

```
{
 SendToSlackWebHook
('https://hooks.slack.com/services/********/*********/*****************',
      'Some message', 'Syncplify.me AFT!', ':smile:');
}
```

Send SMS via Twilio

```
function SendSMSViaTwilio(
      twilioSid,    // string
      twilioToken,  // string
      senderNum,    // string
      recipientNum, // string
      message       // string
)
```

This function sends an SMS (text) message to a recipient (cell)phone number via Twilio.

The `twilioSid` and `twilioToken` parameters are the SID and AuthToken issued to you by Twilio when you signed up for the service.

The `senderNum` is one of the Twilio numbers you've been assigned; this number will be the phone number of the sender of the SMS. The `recipientNum` is the phone number to which you are trying to send the SMS. Both of these numbers shall be in the international phone number standard format (example: `"+15550005555"`).

The `message` parameter is a short string (SMS texts may have a limited length that varies based upon

technology and carrier, typically 140 or 280 characters). This is the actual message that Twilio will try to deliver to the intended recipient.

This function returns **`true`** if the message was successfully accepted for delivery by Twilio, otherwise it returns **`false`**.
Important note: Twilio's acceptance of a message does not imply that the message will be successfully delivered to the recipient. You can track the delivery through your Twilio management console.

Example:

```
{
 SendSMSViaTwilio('********', '*********', '+12345678901', '+15550005555',
'Hello from AFT!');
}
```

Send an email via SMTP

```
function SendMailViaSMTP(srv, port, user, pass, from, to, subj, body, attach)
```

The `SendMailViaSMTP` function sends an email using the designated SMTP server as a relay. All parameters are strings, except for the `port` parameter which is an integer. Note: the `to` parameter may contain multiple recipients, separated by semi-colon (;) as you can see in the example below.

Example:

```
{
  SendMailViaSMTP('smtp.gmail.com', 587, 'me@me.me', GetSecret('smtp pass'),
'me@me.me', 'you@you.com;it@they.com', 'NEW BACKUP UPLOADED!', 'A new backup
has been uploaded!'), '');
}
```

Run a process

```
function Run(commandLine)
```

The `Run` function spawns a process that executes an external program. The executed program can take command line parameters, as shown in the example below.
This function waits for the spawned process to exit, and then returns**`true`** if it ran without errors, or **`false`** if errors occurred.

Example:

```
{
 if (Run('cmd /c "/my_shell_scripts/some_script.bat"')) {
  Log('Batch script ran successfully');
 }
}
```

Run a process asynchronously

```
function RunAsync(commandLine)
```

The `RunAsync` function spawns a process that executes an external program. The executed program can take command line parameters, as shown in the example below.
This function does not wait for the spawned process to exit, and immediately returns**`true`** if the process was started; if the process wasn't started it returns **`false`**.

Example:

```
{
 if (RunAsync('cmd.exe /c "/my_shell_scripts/some_script.bat"')) {
  Log('Batch script started successfully');
 }
}
```

Resize (resample) a JPEG

```
function JPEGResample(imgFile, maxWidth, maxHeight, quality)
```

The `JPEGResample` function resizes a JPEG image (`imgFile`) using the Lanczos3 resampling method to keep the best possible level of detail.
This function also retains the original image's aspect-ratio, and chooses between `maxWidth` and `maxHeight` whichever one would result in a smaller image. If you wish one of these two parameters to be ignored, simply set it to 0 (zero).
The last parameter (`quality`) is a numeric value, between 1 and 100, that indicates the desired quality of the resulting image inversely proportional to its lossy compression; basically an image saved with quality index of 25 will produce a smaller file but lose a lot more detail than an image saved with quality index of 75.

Example:

```
{
 JPEGResample('./self_portrait.jpg', 800, 600, 90);
}
```

Resize (resample) a PNG

```
function PNGResample(imgFile, maxWidth, maxHeight)
```

The `PNGResample` function resizes a PNG image (`imgFile`) using the Lanczos3 resampling method to keep the best possible level of detail.
This function also retains the original image's aspect-ratio, and chooses between `maxWidth` and `maxHeight` whichever one would result in a smaller image. If you wish one of these two parameters to be ignored, simply set it to 0 (zero).

Example:

```
{
 PNGResample('./some_picture.png', 800, 600);
}
```

Extract JPEG metadata

```
function JPEGMetadata(imgFile)
```

This function extract various meta-information (including full EXIF data) about a JPEG image from a file (`imgFile`).

Example:

```
{
 mdata = JPEGMetadata('./self_portrait.jpg');
 Log(JSON.stringify(mdata));
}
```

Produces an output like this:


```
{
  "Exif": {
    "ApertureValue": [
      "149/32"
    ],
    "ColorSpace": [
      1
    ],
    "ComponentsConfiguration": "",
    "CompressedBitsPerPixel": [
      "5/1"
    ],
    "CustomRendered": [
      0
    ],
    "DateTime": "2003:12:14 12:01:44",
    "DateTimeDigitized": "2003:12:14 12:01:44",
    "DateTimeOriginal": "2003:12:14 12:01:44",
    "DigitalZoomRatio": [
      "2272/2272"
    ],
    "ExifIFDPointer": [
      196
    ],
    "ExifVersion": "0220",
    "ExposureBiasValue": [
      "0/3"
    ],
    "ExposureMode": [
      0
    ],
    "ExposureTime": [
      "1/500"
    ],
    "FNumber": [
      "49/10"
    ],
    "FileNumber": [
      1171771
    ],
    "FileSource": "",
    "FirmwareVersion": "Firmware Version 1.10",
    "Flash": [
      24
    ],
    "FlashpixVersion": "0100",
    "FocalLength": [
      2,
      682,
      286,
      215
    ],
    "FocalPlaneResolutionUnit": [
```

```
    2
  ],
  "FocalPlaneXResolution": [
    "2272000/280"
  ],
  "FocalPlaneYResolution": [
    "1704000/210"
  ],
  "ImageType": "IMG:PowerShot S40 JPEG",
  "InteroperabilityIFDPointer": [
    1416
  ],
  "InteroperabilityIndex": "R98",
  "Make": "Canon",
  "MakerNote": "",
  "MaxApertureValue": [
    "194698/65536"
  ],
  "MeteringMode": [
    2
  ],
  "Model": "Canon PowerShot S40",
  "ModelID": [
    17891328
  ],
  "Orientation": [
    1
  ],
  "PixelXDimension": [
    2272
  ],
  "PixelYDimension": [
    1704
  ],
  "ResolutionUnit": [
    2
  ],
  "SceneCaptureType": [
    0
  ],
  "SensingMethod": [
    2
  ],
  "ShutterSpeedValue": [
    "287/32"
  ],
  "ThumbJPEGInterchangeFormat": [
    2036
  ],
  "ThumbJPEGInterchangeFormatLength": [
    5448
  ],
  "UserComment": "",
  "WhiteBalance": [
    0
  ],
  "XResolution": [
    "180/1"
  ],
  "YCbCrPositioning": [
    1
  ],
  "YResolution": [
    "180/1"
  ]
```

```
  },
  "Height": 360,
  "Valid": true,
  "Width": 480
}
```

Extract PNG metadata

```
function PNGMetadata(imgFile)
```

This function extract various meta-information about a PNG image from a file (`imgFile`).

Example:

```
{
 mdata = PNGMetadata('./self_portrait.png');
 Log(JSON.stringify(mdata));
}
```

Produces an output like this:

```
{
  "Height": 360,
  "Valid": true,
  "Width": 480
}
```

Log a custom log line

```
function Log(object)
```

This function adds your own custom line to the script's execution log. Typically you'd call this function with a string parameter, but truly it will log whatever you pass to it.

It's practically very similar to `Log()` and it's used the same way. But our own `Log()` function is to be preferred to Log() because our `Log()` function always works, even for scripts run from within the web interface, whereas Log() only works when you run your scripts from within your operating system's shell.

Example:

```
{
 Log('Hello!!');
 Log(JSON.stringify(some_data));
}
```

Detect halt requests

```
function HaltSignalReceived() // boolean
```

You can check the result of the `HaltSignalReceived()function` anywhere in your script. If it returns **`true`** it means that an administrator (or the Operating System itself) has requested the execution of the current script to be stopped.

Once a halt request is received, the execution of the script will be stopped regardless of whether or not you check this value, but checking it gives you the chance to handle the situation gracefully.

Example:

```
{
 ConsoleFeedback = true;
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'some_username';
 scli.PassFromSecret = 'name_of_the_secret_password';
 if (scli.Connect()) {
  watchr = NewFsWatcher();
  watchr.WatchDir('C:\\Docs', true);
  watchr.Start();
  while (true) {
   Sleep(1000);
   if (HaltSignalReceived()) {
    break; // break infinite loop if halt signal was received
   }
   evt = watchr.Events() // gets the list of pending events to be handled
   if (evt.length > 0) {
    for (var i = 0; i < evt.length; i++) {
     if (evt[i].Event == 'WRITE') {
      scli.UploadWithPath(evt[i].Object, '/realtimebackup', 0);
     }
    }
   }
  }
  scli.Close();
 }
 scli = null;
}
```

Sleep (pause execution)

```
function Sleep(milliseconds)
```

Unlike many other programming languages, JavaScript doesn't really have a native Sleep function (although there are several ways to implement it, for those willing to spend a few lines of code on it). This utility function just makes it much easier to pause the execution of an aftJS script for a certain number of milliseconds, when needed.

Get a secret

```
function GetSecret(secretName)
```

Syncplify.me AFT! allows you to store secrets (strings) in its encrypted database, so you don't have to put them in clear in your scripts. A typical example is a password to a remote file server, you definitely don't want to type such password in plain-text in your script, so you can use`GetSecret` to conceal it.

In Syncplify.me AFT! every secret is identified by another non-secret string, which is the secret's "name" or "description", which you decide when you create and store the secret. Let's say, for example, that your SFTP server's password is "P@ssw0rd", you can store it encrypted in Syncplify.me AFT! and call it "my SFTP password". If you do so, then you can write your script like this:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.Password = GetSecret('my SFTP password');
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
```

```
 }
 scli = null
}
```

This is a very context-agnostic way to store and use any type of secret in Syncplify.me AFT!... for passwords specifically, don't forget that most client objects provide the PassFromSecret property, which can be used like this:

```
{
 var scli = new SftpClient();
 scli.Host = 'your.sftpserver.com:22';
 scli.User = 'someusername';
 scli.PassFromSecret = 'my SFTP password';
 if (scli.Connect()) {
  // perform your file transfers...
  // ...
  // ...
  scli.Close();
 }
 scli = null
}
```

Extract file path

```
function ExtractPath(fullyQualifiedFileName)
```

This function takes a fully qualified path name (root-based path including all directories and the file name) and returns the path portion only.

Example:

```
{
 res = ExtractPath('/docs/sheets/budget.xlsx'); // res will be "/docs/sheets"
}
```

Extract file name

```
function ExtractName(fullyQualifiedFileName)
```

This function takes a fully qualified path name (root-based path including all directories and the file name) and returns the file-name portion only.

Example:

```
{
 res = ExtractName('/docs/sheets/budget.xlsx'); // res will be "budget.xlsx"
}
```

Extract file extension

```
function ExtractExt(fullyQualifiedFileName)
```

This function takes a fully qualified path name (root-based path including all directories and the file name) and returns the file extension portion only.

Example:

```
{
```

```
 res = ExtractExt('/docs/sheets/budget.xlsx'); // res will be ".xlsx"
}
```

Number to string (with padding)

```
function NumToStrPad(number, length)
```

This functions takes a number and returns a string representation of that number padded with zeroes to ensure that the resulting string is not shorter than `length`. Useful, for example, when you want a 2-digit representation of month and day number in a date.

Example:

```
{
 var date = new Date();
 var month = date.getMonth()+1;
 monthStr = NumToStrPad(month, 2); // Ex: in May, monthStr will contain the
string "05"
}
```

Unique IDs (UUID)

Syncplify.me AFT! provides 3 different functions to create unique IDs according to your taste and preferences.

```
var uid = ShortUID();
// uid will contain a string like Rp68VrvHMNcc5jffKdQaWZ

var uid = LongUID();
// uid will contain a string like oUhCu6wp6DrmyVxVaEHjzRvv3mh3PGNksSUngbX6rd6Q

var uid = UUIDv4();
// uid will contain a string like 0d2b0018-cc08-44b3-bdc1-401b2819cef1
```

Generate a PGP key-pair

```
function GeneratePGPKeys(
  keyPairName, // string
  directory,   // string
  bits         // integer (512, 1024, 2048, ...)
)
```

This function generates a PGP key-pair (public and private keys), and saves both keys as files in the specified `directory`.

The name of both files will be `keyPairName`; the public key's file name will be `kyePairName`**.pubkey**, while the private key's file name will be `keyPairName`**.privkey**.

The `bits` parameter is an integer number, and it must be a PGP-compatible key size; typically these are powers of 2, like 512, 1024, or 2048.

If this function succeeds, it returns **`true`**, otherwise it returns **`false`**.

Example:

```
{
 GeneratePGPKeys('testkey', 'C:\\PGPKeys', 2048);
}
```

Encrypt a file with PGP

```
function PGPEncryptFile(
  inFile,  // string
  outFile, // string
  pubKey,  // string
  privKey  // string
)
```

This function encrypts a file using OpenPGP, the meaning of each parameter is as follows:

- `inFile`: this is the full path and name to the file that you wish to encrypt
- `outFile`: this is the full path and name of the resulting encrypted file you wish to generate
- `pubKey`: the full path and name to a file containing the **recipient**'s PGP public key
- `privKey`: the full path and name to a file containing the **sender**'s PGP private key

If this function succeeds, it returns `true`, otherwise it returns `false`.

Example:

```
{
 PGPEncryptFile('C:\\Data\\budget.xlsx', 'C:\\Encrypted\\budget.xlsx.pgp',
   'C:\\PGPKeys\\Bob.pubkey', 'C:\\PGPKeys\Alice.privkey');
}
```

Decrypt a file with PGP

```
function PGPDecryptFile(
  inFile,  // string
  outFile, // string
  pubKey,  // string
  privKey  // string
)
```

This function decrypts a file using OpenPGP, the meaning of each parameter is as follows:

- `inFile`: this is the full path and name to the PGP-encrypted file that you wish to decrypt
- `outFile`: this is the full path and name of the unencrypted/plain resulting file you wish to generate
- `pubKey`: the full path and name to a file containing the **recipient**'s PGP public key
- `privKey`: the full path and name to a file containing the **recipient**'s PGP private key

If this function succeeds, it returns `true`, otherwise it returns `false`.

Example:

```
{
 PGPDecryptFile('C:\\Received\\budget.xlsx.pgp', 'C:\\Data\\budget.xlsx',
   'C:\\PGPKeys\\Bob.pubkey', 'C:\\PGPKeys\Bob.privkey');
}
```

Why adding 3rd party stuff?

JavaScript is a great language, very easy to learn ad use on a daily basis.
But by itself it lacks a few features that would make programmers' lives a lot easier.

The first thing is the beloved `fetch` function that's available to JavaScript when running inside a browser, like Chrome or FireFox. Since our environment is not a browser, we had to bake the`fetch` function into our aftJS language ourselves, and we did.

Another well-known and loved JavaScript library that's used by tons of programmers isUnderscore.js so we added that one in as well.

How to "require" a Node.js/JavaScript module

Similarly (yet not identically) to Node.js, the aftJS language supports the`require` keyword. Requiring a module is necessary in order to use any of its exported functions. Module functions cannot be used if the module containing them hasn't been required first.

For example, this script would fail/crash at runtime:

```
{
 // Will crash because the underscore module hasn't been required
 if (_.contains([1, 2, 3], 3)) {
  Log('Yay!');
 }
}
```

But this script would compile and run flawlessly:

```
{
 // Require the minified Underscore.js module
 var _ = require("underscore-min");
 // Let's use the "contains" function from the previously required
Underscore.js module
 if (_.contains([1, 2, 3], 3)) {
  Log('Yay!');
 }
}
```

In order to require modules, they have to be installed in the "modules" subdirectory of AFT!'s configuration folder. Typically this folder is located:

- In Windows: *C:\ProgramData\Syncplify.me\AFTv1\modules*
- In Linux (and other Posix OSs): */etc/xdg/Syncplify.me/AFTv1/modules*

IMPORTANT NOTE: AFT! does **not** use nor integrate Node.js; although we try our best to ensure compatibility with Node.js modules, we cannot guarantee that all Node.js modules will work in the AFT! runtime environment.

The famous "underscore.js" library

Underscore is a JavaScript library that provides a whole mess of useful functional programming helpers without extending any built-in objects. It’s the answer to the question: “If I sit down in front of a blank HTML page, and want to start being productive immediately, what do I need?”

Underscore provides over 100 functions that support both your favorite workaday functional helpers: **map, filter, invoke** — as well as more specialized goodies: function binding, javascript templating, creating quick indexes, deep equality testing, and so on.

Learn more about it on the official Underscore.js web site.

Syncplify.me AFT! comes equipped out-of-the-box with the minified version of the underscore.js module. In order to use it you'll need to require the module.

Example:

```
{
 // Require the minified Underscore.js module
 var _ = require("underscore-min");
 // Let's use the "contains" function from the previously required
Underscore.js module
 if (_.contains([1, 2, 3], 3)) {
  Log('Yay!');
 }
}
```