Monitoring a local folder and uploading files to your SFTP server when they change

> This article refers to **AFT! v4.0.0+**, if you're still on an older AFT! version, please, refer to this article instead.

This article explains how to implement real-time folder backup using Syncplify AFT! scripting. Two approaches are presented: one optimized for low-frequency changes, and one for high-frequency changes. Both use `FsWatcher` to monitor a local folder and `SftpClient` to upload changed files to a remote SFTP server as soon as they appear.

How It Works

AFT! scripts run as long-lived jobs. Rather than executing once and exiting, they loop indefinitely by calling `WaitForHaltSignal()`, which blocks the script until an operator stops the job from the web UI or via the API. This makes them ideal for reactive, always-on automation tasks like real-time backup.

The `FsWatcher` object monitors a local directory (and optionally its subdirectories) for filesystem events. When an event occurs, it invokes a callback function with an event object containing two key fields:

- `evt.Event`: the type of event (one of `EVT_CREATE`, `EVT_WRITE`, `EVT_REMOVE`, `EVT_RENAME`, `EVT_CHMOD`)
- `evt.Object`: the full path of the affected file or directory

The script filters the events it cares about and takes action accordingly.

Choosing The Right Approach

The two strategies differ in how they manage the SFTP connection lifecycle:

	Connect On Demand	**Persistent Connection**
SFTP connection	Opened and closed per event	Opened once at startup

Best for	Infrequent changes	Frequent changes
Risk	Higher latency per upload	Idle timeout disconnections

If files in the watched folder change many times per minute, opening and closing a TCP connection for each event adds unnecessary overhead. In that case, a persistent connection is the better choice. Conversely, if files change rarely (for example, a few times per hour), keeping a connection open indefinitely risks hitting the server's idle timeout. Opening on demand avoids that entirely.

Script 1: Connect on Demand

In this approach, the SFTP client is created fresh inside the event callback. Each detected change results in a new connection being opened, the file uploaded, and the connection immediately closed.

```
// Real-time backup: CONNECT ON DEMAND
// Suitable for folders where files change very infrequently. A fresh SFTP connection
// is opened and closed for each detected change, avoiding idle connection timeouts.

// Configure which filesystem events trigger a backup upload.
var watcher = new FsWatcher();
watcher.NotifyCreate = true;
watcher.NotifyWrite  = true;
watcher.NotifyRemove = false;
watcher.NotifyRename = false;
watcher.NotifyChmod  = false;

try {
    // Watch the source folder recursively (true = include all subdirectories).
    watcher.WatchDir("C:\\YourSourceFolder", true);
    watcher.Start(function(evt) {
        // Ignore events we don't care about.
        if (evt.Event !== EVT_CREATE && evt.Event !== EVT_WRITE) {
            return;
        }

        Log("change detected: " + evt.Object + ", connecting to SFTP");

        // Open a fresh connection for each event, upload the changed file, then close.
        var scli = new SftpClient();
        scli.Host = "your-sftp-server.example.com:22";
        scli.Username = "your-username";
```

```
        scli.Password = GetSecret("your-password-secret-name");
        scli.HostKeySHA256 = "your-servers-public-key-sha256-fingerprint";

        if (scli.Connect()) {
            // UploadWithPath preserves the relative path structure on the remote side.
            scli.UploadWithPath(evt.Object, "/remote/backup/path", 0);
            scli.Close();
        } else {
            Log.Error("could not connect to SFTP server to upload: " + evt.Object);
        }
        scli = null;
    });

    // Block here until an operator sends a halt signal via the AFT! web UI or API.
    WaitForHaltSignal();
    Log("halt signal received, shutting down");
} finally {
    // Always stop the watcher on exit, whether normal or due to an error.
    watcher.Stop();
}
```

A few things worth noting:

`watcher.WatchDir("C:\\YourSourceFolder", true)`: the second argument (**true**) enables recursive watching. Any file created or modified inside a subdirectory, at any depth, will trigger the callback. AFT!'s `FsWatcher` also automatically tracks newly created subdirectories, so folders that did not exist when the watcher started are covered without any extra logic.

`UploadWithPath(evt.Object, "/remote/backup/path", 0)`: unlike a plain `Upload`, `UploadWithPath` preserves the relative directory structure of the source file when writing it to the remote destination. The third argument is the number of leading path components to strip. Passing 0 means the full path (minus the drive letter on Windows) is reconstructed on the remote side, maintaining a mirror of the local structure.

`scli.HostKeySHA256`: this is the SHA-256 fingerprint of the SFTP server's public host key. Setting it enables strict host key verification, which prevents man-in-the-middle attacks. The fingerprint can be obtained from your SFTP server administrator or by inspecting the server's known host key. Omitting it is not recommended in production environments.

The `try/finally` block — the watcher is started inside `try` and always stopped in `finally`. This guarantees that system resources (file handles and OS-level filesystem notification handles) are released even if the script exits due to an error.

Script 2: Persistent Connection

In this approach, the SFTP client is created and connected before the watcher starts. The same connection is reused for every upload event throughout the lifetime of the job.

```
// Real-time backup: PERSISTENT SFTP connection
// Suitable for folders where files change frequently. The SFTP connection is opened
// once at startup and kept alive by the constant upload activity.
// Connect first, then start watching. On halt, stop the watcher and close the connection.

// Set up and connect the SFTP client before starting the watcher.
var scli = new SftpClient();
scli.Host = "your-sftp-server.example.com:22";
scli.Username = "your-username";
scli.Password = GetSecret("your-password-secret-name");
scli.HostKeySHA256 = "your-servers-public-key-sha256-fingerprint";

// Configure which filesystem events trigger a backup upload.
var watcher = new FsWatcher();
watcher.NotifyCreate = true;
watcher.NotifyWrite  = true;
watcher.NotifyRemove = false;
watcher.NotifyRename = false;
watcher.NotifyChmod  = false;

// Fail fast: abort if the server is unreachable at startup.
if (!scli.Connect()) {
    Log.Error("could not connect to SFTP server, aborting");
    Exit();
}

try {
    // Watch the source folder recursively (true = include all subdirectories).
    watcher.WatchDir("C:\\YourSourceFolder", true);
    watcher.Start(function(evt) {
        if (evt.Event === EVT_CREATE || evt.Event === EVT_WRITE) {
            Log("uploading: " + evt.Object);
            // UploadWithPath preserves the relative path structure on the remote side.
            scli.UploadWithPath(evt.Object, "/remote/backup/path", 0);
        }
```

```
    });

    // Block here until an operator sends a halt signal via the AFT! web UI or API.
    WaitForHaltSignal();
    Log("halt signal received, shutting down");
} finally {
    // Always clean up both the watcher and the connection on exit.
    watcher.Stop();
    scli.Close();
}
```

The structure differs from Script 1 in a few important ways:

Connect before watching: the `scli.Connect()` call happens at the top level, before `WatchDir` or `Start`. If the connection fails at startup, the script calls `Exit()` immediately. There is no point starting the watcher if there is nowhere to send the files.

No per-event connection setup: the callback is intentionally minimal. It logs the filename and calls `UploadWithPath`. The absence of connect/close calls inside the callback is not an oversight; it is the entire point of this pattern.

`finally` **closes both:** in this version, the `finally` block must stop the watcher and close the SFTP connection. The watcher must be stopped first so no new uploads are attempted after the connection is gone.

When to Use Each Pattern

Use Connect on Demand when:

- The watched folder receives new or modified files at most a few times per hour
- The remote SFTP server enforces a short idle connection timeout
- You prefer simplicity and do not want to think about connection state

Use Persistent Connection when:

- Files change frequently (multiple times per minute or more)
- Upload latency matters and you want to avoid the TCP handshake overhead per event
- The remote server's idle timeout is long enough, or keep-alives are configured

Both patterns are valid. The choice is an operational one based on your environment.

A VFS-based alternative

For backup purposes like the ones shown in this article, using a Remote Client Object (as seen here above) is oftentimes the best option, because of the inherent flexibility and advanced configuration

options. After all, Remote Client Objects are purpose-built exactly to carry out these tasks.

But if you need a quick way to achieve the same goal, and you don't need advanced features like atomic uploads, `chtimes` after successful uploads, rule-based skipping, etc, you may prefer the simplicity of a VFS-based approach. Virtual File Systems, in fact, manage the state of their remote connection automatically and internally, hiding all that complexity, and providing an even easier-to-use paradigm.

Here's a version of the above scripts rewritten to use a VFS instead of a Remote Client Object. Note: the named VFS configuration must exist in AFT! before running this script.

```
// Real-time backup using VirtualFS
// Because VirtualFS manages the underlying connection automatically (reconnecting
// as needed), a single script covers both the "infrequent changes" and "frequent
// changes" scenarios with no manual connect/close logic anywhere.
//
// The preferred approach is VirtualFSByName, which resolves credentials from the
// VFS Library so no passwords ever appear in script source. Alternatively, construct
// the VFS inline (see the commented example further below).

var sourceFolder = "C:\\YourSourceFolder";
var remoteFolder = "/remote/backup/path";

// Configure which filesystem events trigger a backup upload.
var watcher = new FsWatcher();
watcher.NotifyCreate = true;
watcher.NotifyWrite  = true;
watcher.NotifyRemove = false;
watcher.NotifyRename = false;
watcher.NotifyChmod  = false;

// Resolve the SFTP connection from the VFS Library.
// Credentials, host key, and all options are stored in the named profile.
var sftp = new VirtualFSByName("your-sftp-vfs-name");

// Inline alternative, if you prefer to specify the target directly in the script:
//   var sftp = new VirtualFS(VfsTypeSFTP, "sftp://username:password@your-sftp-server.example.com:22", "");

try {
    watcher.WatchDir(sourceFolder, true);
    watcher.Start(function(evt) {
```

```
        if (evt.Event !== EVT_CREATE && evt.Event !== EVT_WRITE) {
            return;
        }

        // Compute the remote path by stripping the local source root prefix and
        // normalising backslashes to forward slashes for the remote side.
        var relativePath = evt.Object.substring(sourceFolder.length).replace(/\\/g, "/");
        var targetPath   = remoteFolder + relativePath;
        Log("uploading: " + evt.Object + " -> " + targetPath);

        // Use ImportFile, which automatically ensures that target directory-tree
        // is created if it doesn't already exist
        var r = sftp.ImportFile(evt.Object, targetPath);
        if (!r.Ok()) {
            Log.Error("upload failed for " + evt.Object + ": " + r.ErrorMsg());
        }
    });

    // Block here until an operator sends a halt signal via the AFT! web UI or API.
    WaitForHaltSignal();
    Log("halt signal received, shutting down");
} finally {
    // Always stop the watcher on exit, whether normal or due to an error.
    watcher.Stop();
}
```

Now you know all the options at your disposal. The choice is yours.

Revision #4
Created 10 April 2026 13:03:17 by DevTeam
Updated 12 April 2026 01:35:32 by DevTeam