Integrating AFT! with an external workflow API

Many serious file transfer operations are parts of larger business workflows. A data pipeline might require a "claim" step where the downstream system registers its intent to process a batch before any files move, and a "completion" step where status is reported back once the transfer finishes, whether successfully or not. AFT! can play the role of execution engine in those orchestration patterns by making HTTP calls before and after the actual file movement. This article shows how to build a script that claims a job from an external API, performs the transfer, and always reports the outcome back to that same API, even if the transfer fails.

How It Works

The script receives a job identifier as a runtime parameter so the same script definition can be reused across multiple jobs without modification. It calls a claim endpoint on the orchestration API to retrieve the VFS names and paths to use for this particular run. It then performs the transfer and calls a report endpoint with the outcome.

```
// REST API integration: claim, transfer, report
// Receives a job ID as a runtime parameter, claims the job from an external
// orchestration API to discover source and destination details, performs the
// file transfer, and always reports the outcome back to the API.

var jobId       = Params("job_id");      // passed in when triggering the job
var apiBase     = GetSecret("orch-api-url");
var apiToken    = GetSecret("orch-api-token");

if (!jobId) {
    Log.Error("job_id parameter is required");
    Exit(1);
}

// Step 1: Claim the job from the orchestration API.
// This tells the API that AFT! is about to start work and retrieves the
// source/destination details for this specific run.
var claimResp = HttpCli()
```

```
    .Url(apiBase + "/jobs/" + jobId + "/claim")
    .Bearer(apiToken)
    .Post(null);

if (!claimResp.Ok()) {
    Log.Error("claim failed: HTTP " + claimResp.StatusCode());
    Exit(1);
}

var job = JSON.parse(claimResp.BodyAsString());
var transferOk = false;

// Step 2: Perform the transfer.
try {
    var src = VirtualFSByName(job.sourceVfs);
    var dst = VirtualFSByName(job.destVfs);

    var cpResult = src.CopyToVFS(job.sourcePath, dst, job.destDir);
    if (!cpResult.Ok()) {
        Log.Error("transfer failed: " + cpResult.ErrorMsg());
    } else {
        Log.Info("transfer complete: " + job.sourcePath + " -> " + job.destVfs + job.destDir);
        transferOk = true;
    }
} catch (err) {
    Log.Error("transfer exception: " + err.toString());
}

// Step 3: Always report the outcome, success or failure.
// The orchestration system needs to know what happened regardless of how
// the transfer ended so it can route the workflow correctly.
var reportPayload = JSON.stringify({
    jobId:   jobId,
    status:  transferOk ? "completed" : "failed",
    agent:   "aft"
});

var reportResp = HttpCli()
    .Url(apiBase + "/jobs/" + jobId + "/report")
    .Bearer(apiToken)
```

```
    .Header("Content-Type", "application/json")
    .Post(reportPayload);

if (!reportResp.Ok()) {
    Log.Error("report endpoint returned HTTP " + reportResp.StatusCode());
    // Do not re-Exit here; fall through to the final exit below with the right code.
}

if (!transferOk) {
    Exit(1);
}
```

`Params("job_id")`: the `Params` function retrieves values passed to the job at trigger time. When you trigger this script from the AFT! API or from the CLI, you supply the parameter as a key/value pair. The script logic stays generic; the job-specific data comes in at runtime.

`GetSecret("orch-api-token")`: storing the bearer token as a named secret keeps credentials out of the script source entirely. Rotate the secret in the AFT! secrets store and every script that calls `GetSecret` automatically picks up the new value without any changes to the script.

`claimResp.BodyAsString()`: returns the raw response body as a string. Wrap it in `JSON.parse()` to work with the object. `HttpCli` does not parse JSON automatically; the choice of which body fields to use is left to the script.

The report call sits outside the `try/catch` and sends regardless of outcome. This is deliberately unconditional. If you only report on success or only on error, the orchestration system can end up waiting indefinitely for a status it will never receive, which is worse than almost any other failure mode.

Triggering the script with parameters from the CLI

```
aft start --scriptid <your-script-id> --apikey <your-api-key> --params '[{"name":"job_id","value":"abc-123"}]'
```

Params are a JSON array of `{name, value}` objects. The same format is accepted by the `POST /v1/jobs` REST endpoint, so any HTTP client or orchestration system can trigger AFT! jobs and pass context in directly.

Triggering from the REST API with curl

```
curl -s -X POST https://127.0.0.1:44399/v1/jobs \
  -H "X-API-Key: your-api-key" \
  -H "Content-Type: application/json" \
  -d '{"scriptId":"your-script-id","params":[{"name":"job_id","value":"abc-123"}]}'
```

The job starts immediately and responds with a job ID you can use to poll the status endpoint.

Revision #1
Created 12 April 2026 15:03:26 by DevTeam
Updated 12 April 2026 15:25:43 by DevTeam