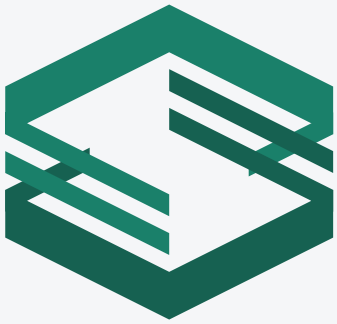
Manual 11 · flows that wait for events: webhooks, watchers, watchdogs

Services



Most flows start, process what is there and end. A service waits instead — for files in a folder, calls over HTTP, messages from a broker, a green light from another system — and reacts each time. This manual explains how services run in Stilton, the nodes that make them (Dir Watcher, Webhook Receiver, Webhook Responder, Watchdog, message sources), how they stop, what is and is not guaranteed, and how to run them on a server.

CONTENTS

01 Batch jobs and services

02 The service nodes

- 2.1 Dir Watcher — files in a folder
- 2.2 Webhook Receiver — calls over HTTP
- 2.3 Webhook Responder — answers GET and HEAD
- 2.4 Watchdog — waits for another system
- 2.5 MQTT and ActiveMQ sources

03 Stopping

04 Failures

05 Transactions in services

06 Security

07 Running a service on a server

08 Patterns

09 Limits

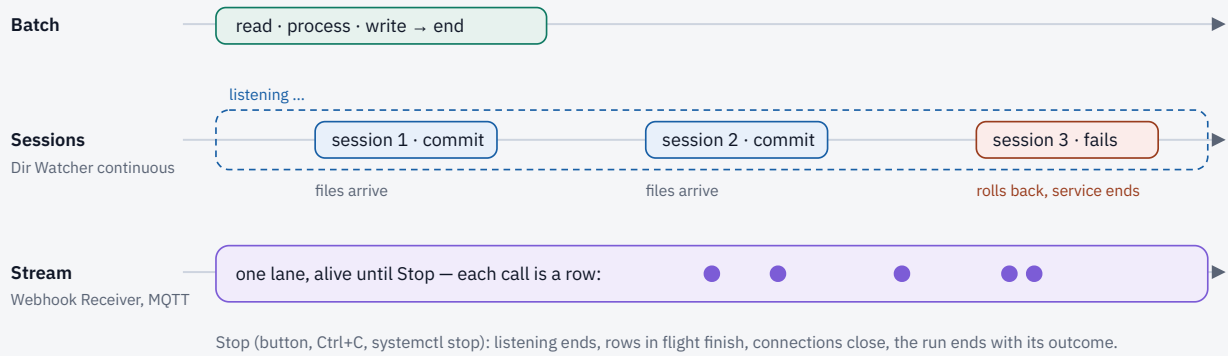
Copyright © 2025–2026 Marco Dongu. All rights reserved.

01 Batch jobs and services

A **batch job** has a beginning and an end: it reads what is there, processes it and stops. A scheduler starts it at a set time and reads its exit code.

A **service** does not end by itself. One of its nodes **listens**, and the run stays alive until someone stops it. Stilton has three ways of running a service, and the difference matters for transactions, failures and memory:

Way	Nodes	How the lane runs
Sessions	Dir Watcher *Watch* → *Continuous*, FTP/SFTP Source in continuous mode	the lane runs **once per group of events**, from start to end; each run is a *session*
Stream	Webhook Receiver, MQTT Source, ActiveMQ Source, Watchdog *Stream* / *Edge*	the lane runs **once** for the whole life of the service; rows flow through it as they arrive
Gate	Watchdog *Gate*, Dir Watcher *Watch* → *Once*	the flow waits for **one** event, then goes on and ends like a batch job



The same Stop ends all three. What differs is how often the lane runs, and therefore what a transaction and a failure cover.

Choosing. Files are best handled in sessions: each group of files is complete in itself, can be committed or rolled back as a whole, and a failed file is picked up again at the next start. Calls and messages are best handled as a stream: each row stands alone and is written at once, in autocommit, with an UPSERT so that a repeated event does no harm.

02 The service nodes

Dir Watcher — files in a folder

Emits one row per file (`path` , `filename` , `size` , `modified_at` , `event` …) of a **local** folder.

Option	Values	Notes
Mode	*Scan* · *Watch*	Scan lists what is there now; Watch listens to the operating system

Option	Values	Notes
Listening (Watch)	*Once · Continuous (per session)*	Once waits for the first group and ends; Continuous runs a session per group
Events	all · new files only · …	with Continuous choose **New files only**: copying a file produces several events
Pattern, subfolders	`*.csv`, `invoice_*.xml`	events for other files never start a session
File stability	seconds	a file is emitted only when its size and date stop changing
De-duplication	path · path + date · content (SHA-256)	with a **state file** it survives restarts

A group of events opens a session only if it contains at least one event the node would emit. Late events, such as the closing of a copied file or the de-duplication state file being written, do not re-run the lane. A **File Input** with *Path source = flow* reads one file per row it receives, so a Dir Watcher feeds it directly.

Example. Watch `inbox`, *Continuous*, *New files only*, pattern `*.csv` → File Input (path from the flow) → transform → DB Sink in a transaction. Two files dropped together are one session and one transaction. A broken file rolls back its own session, ends the service, and is not marked as processed: after the fix it is loaded at the next start. *(Guide 11.)*

On mounted network folders the operating system often sends no events: use **Scan** with de-duplication and a state file, started by the scheduler every few minutes.

Webhook Receiver — calls over HTTP

Listens on a port and a path. Each call becomes a row with the fixed fields below, plus the fields of a JSON body. The caller is answered **at once**.

Field	Content
`event_id`	the delivery header (`X-Webhook-Delivery`, `X-Delivery-Id`, `X-GitHub-Delivery`, `X-Shopify-Webhook-Id`), or else a fingerprint of the body
`event_type`	`X-Webhook-Event`, `X-GitHub-Event` or `X-Shopify-Topic`
`payload`, `headers`	the body and the headers; read a key with `get(payload, "order_id")`
`source_ip`, `webhook_path`, `received_at`, `signature_valid`	where, when, and the result of the HMAC check (null without a secret)

Answer	When
`200 {"status":"received"}`	the call was accepted (it becomes a row, or it was a duplicate)
`401 {"status":"invalid_signature"}`	a secret is set and the signature is missing or wrong — no row
`503 {"status":"queue_full"}` + `Retry-After: 5`	the buffer is full and its policy is *refuse* — the caller should retry
no answer	the caller's address is not in the IP allowlist

Accepted is not processed. The Receiver answers before the flow has handled the row. A problem found later, such as a failed check or a database refusing the row, goes to the log, the reject port or the Error Handler, as for any row. If the caller must know the outcome, give it an endpoint to ask (Responder, below) or call it back.

Duplicates. Within the de-duplication window (default one hour) an `event_id` seen before is answered 200 and dropped. Without a delivery header the id is a fingerprint of the body, so the same body sent twice counts once. That is what you want when a sender retries, but two genuinely identical calls also count once: include an id or a timestamp in the body.

Buffer. Calls wait in a buffer (default 1 000) while the flow catches up. When it is full: *drop oldest*, *drop newest*, or *refuse with 503*. Choose 503 when nothing may be lost: well-behaved senders retry.

Several Receivers can share a port with different paths.

Webhook Responder — answers GET and HEAD

Answers `HEAD` and/or `GET` on a port and a path with **headers you define**. Each header is a fixed value or a `$name`:

- **Flow** mode — `$name` is a field of the latest row that reached it; the rows pass on unchanged. A status endpoint: *"the last order received was O-802". (Guide 15.)*
- **Monitor** mode — `$name` is a lane variable; no input. A fixed signal: *"data ready: true"*.

`HEAD` returns the headers with `204 No Content`. `GET` returns `200` with the headers, also as JSON in the body unless *Body on GET* is off. **Exposure time** limits how long it answers (0 = until stopped).

Watchdog — waits for another system

Calls a URL every *interval* seconds (`HEAD` or `GET`, with optional Bearer, Basic or API-key authentication) and looks at **one header**: exact value, contains, or simply present.

Mode	Behaviour	Ends
Gate	holds the rows it receives (or emits one row) until the condition is true, then lets them through with `watchdog_matched`, `watchdog_attempts`, `watchdog_elapsed_ms` …	after it opens, or at the limit: *Max attempts* / *Global timeout* → error or proceed with `watchdog_matched = false`
Stream	a row at every positive check	when stopped, or after *Duration*
Edge	a row only when the state changes, with `watchdog_edge` = `rising` / `falling`	when stopped, or after *Duration*

A connection error is not a failure: the check counts as negative and the Watchdog tries again.

Example — wait for the upstream. File Input → Watchdog *Gate* on `https://shop.example/api/status`, header `X-Day-Closed: true`, every 5 minutes, global timeout 120 minutes, *Error* at timeout → DB Sink. Started at 23:00 by the scheduler, the load runs as soon as the shop closes its day. If the shop never does, it fails at 01:00 with exit code 1, and the scheduler alerts someone. *(Guide 16.)*

Example — a sentinel. Watchdog *Edge*, *Rising and falling*, on the health endpoint of a partner API every 30 seconds → Mail Sink "partner API: {watchdog_edge}". Running as a service, it mails once when the API goes down and once when it comes back.

MQTT and ActiveMQ sources

Subscribe to a broker and emit one row per message, as a stream. MQTT 3.1.1 with topic wildcards, QoS and TLS; ActiveMQ over STOMP, queue or topic, with durable subscriptions that keep the messages sent while the flow is stopped. Their options are in the node notes (manual 08). The rules of this manual for streams apply to them unchanged.

03 Stopping

The same request stops every service:

Where	How
studio	**Stop**
terminal	Ctrl+C
Linux service	`systemctl stop` (SIGTERM)
Windows	*End* in Task Scheduler
inside the flow	a **Stop** node — ends the lane on purpose, with your message

A first request is **clean**:

01 listening nodes stop waiting;

02 the rows already in flight finish their way through the lane;

03 sessions in progress commit;

04 connections close;

05 the run ends with its outcome.

A **second** request exits at once with code `130`: the rows in flight are lost and the database rolls back open transactions. Give a service enough time to stop: on Linux `TimeoutStopSec=120` is a reasonable start.

04 Failures

In a session (Dir Watcher, FTP continuous): the failing session rolls back its transaction, the sessions before it stay committed, and **the service ends** with exit code 1. That is deliberate. The file that broke stays where it is, not marked as processed, and a service manager restarts the runner (`Restart=on-failure`). A file that fails every time would make the service restart forever, so move it away as soon as the alert arrives.

In a stream (Receiver, MQTT, ActiveMQ): a failing row follows the normal rules of the lane: the reject port of the node, its catch port, the Error Handler. With the default rule (*Always → Emit*) the error is logged and sent on `error_out`, and the service keeps listening. The service ends only when the Error Handler turns the error into a stop: the node is marked **critical**, or a rule says **Stop the lane**. Do that only for what makes it pointless to go on, such as the database being unreachable, and let single bad rows go to a reject file.

Of the listener itself, for example a port already in use or a folder that does not exist: the node fails at start and the run ends immediately with the reason.

05 Transactions in services

- **Sessions**: each session is its own transaction. This is the natural home of transactions in services.
- **Streams**: do not put DB Sinks of a stream lane in a transaction. It would stay open for the whole life of the service, holding locks, and commit or roll back everything at once. Write in autocommit, one event at a time, with an UPSERT keyed on the event (`event_id` or a business key), so that a repeated delivery is harmless.

Manual 10, *Transactions*, has the details.

06 Security

A listening node opens a door on the run machine. Close it as far as the job allows:

Measure	For	How
HMAC signature	Receiver	a shared secret, written as `${SECRET_NAME}` (never in clear); unsigned or wrongly signed calls get 401
IP allowlist	Receiver	only the addresses of the systems that call
TLS	Receiver, Responder	a reverse proxy (nginx, Caddy, IIS) in front of the port: the nodes speak plain HTTP
Bind and firewall	all	open the port only towards the proxy or the callers
Authentication	Watchdog	Bearer, Basic or API key, from a secret

Measure	For	How
Least privilege	all	run the service under a dedicated account that can read only its folders

A Responder publishes what you put in its headers. Expose states and identifiers, never personal data or secrets.

---

07 Running a service on a server

Compile the project into an artifact (manual 03), trust the key on the server, then start the runner **as a service**, not on a schedule. Manual 04 covers admission, configuration and logs; here only what is specific to services.

Linux — /etc/systemd/system/stilton-orders.service — ini

```ini
[Unit]
Description=Stilton service: orders API
After=network-online.target

[Service]
User=stilton
WorkingDirectory=/opt/stilton/jobs/orders
EnvironmentFile=/etc/stilton/orders.env
ExecStart=/usr/local/bin/stilton-runner /opt/stilton/jobs/orders/orders-api.ffart
Restart=on-failure
RestartSec=30
TimeoutStopSec=120
StandardOutput=append:/var/log/stilton/orders-api.ndjson

[Install]
WantedBy=multi-user.target
```

`orders.env` holds the secrets and the monitor address (`ORDERS_HOOK_SECRET=…`, `MONITOR_URL=…`, `MONITOR_TOKEN=…`), readable only by the `stilton` account.

On **Windows**, create a task in Task Scheduler:

- **Trigger**: *At startup*;
- **Account**: the service account, *Run whether user is logged on or not*;
- **Action**: `stilton-runner.exe` with the artifact as argument and the job folder as *Start in*;
- **Settings**: *If the task fails, restart every 1 minute* and *Do not start a new instance*.

Send its events to Stilton Monitor (manual 06). A service that has been listening for days shows a long run whose events tell you what it received, and a failure shows the reason in the run summary.

08 Patterns

You need to…	Build	Guide
load files dropped into a folder, as they arrive	Dir Watcher *Continuous* → File Input (path from flow) → … → DB Sink in a transaction	11
accept records pushed over HTTP	Webhook Receiver → Transform `get(payload, …)` → Filter → sinks	15
publish the state of a flow to other systems	… → Webhook Responder *Flow* (or *Monitor* with lane variables)	15
start a load when another system is ready	File Input → Watchdog *Gate* → sinks, started by the scheduler	16
be told when a service goes down or comes back	Watchdog *Edge* → Log / Mail Sink, as a service	16
consume a message queue	ActiveMQ / MQTT Source → … → DB Sink with UPSERT	—

09 Limits

- One run machine, one process: a service does not spread its work over several machines. Run more copies on different ports or folders, or wait for the orchestrator of version 3.0.
- The Receiver keeps its buffer and its de-duplication window **in memory**: a restart forgets them. Senders that retry, and an UPSERT, cover the gap.
- Dir Watcher *Watch* needs a local file system. On network shares use *Scan* on a schedule.
- The Responder answers `HEAD` and `GET` only. It is a status endpoint, not a general API.