K2 Alpinist

Course 10

Jobs

WELCOME

Agenda

- Job Types
- Job commands
- Monitoring jobs
- Session Scope
- Jobs Mechanism
  - K2_JOBS table
  - Job’s Life Cycle
  - Managing Job Execution
- Job Implementation
- Heartbeat
- Jobs & Project deployment
- Jobs Actors
- Jobs Configurations
- JMX Stats

Fabric Jobs

Fabric Jobs is a mechanism for running scripts or executables. Once configured, Fabric creates asynchronous tasks (running threads) that execute specific commands, Broadway flows, or Java code at scheduled dates and times.

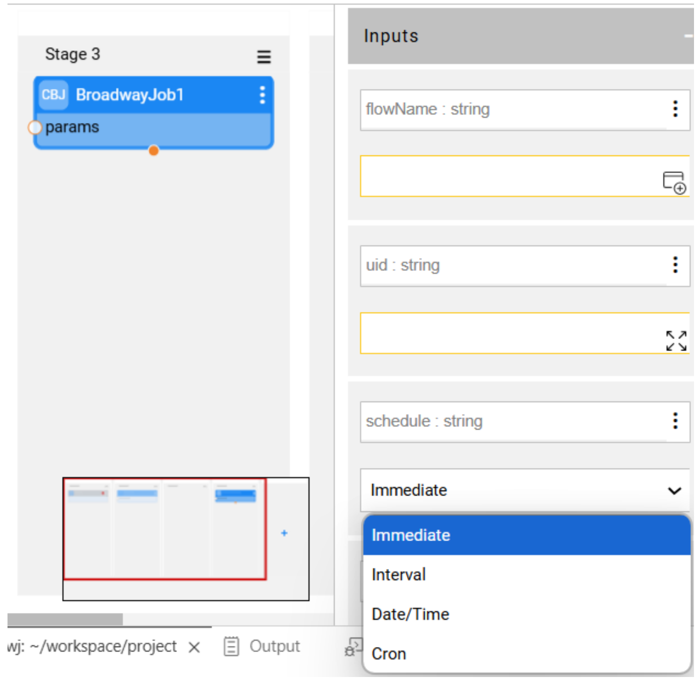
Job types:

- Broadway job
- User job
- Process job
- Interface listener job
- Batch job
- Common tables job
- CDC jobs

Job Types

Broadway job

A Broadway flow defined in the Broadway GUI.
Can be executed using the **BroadwayJob Actor** or the **startJob** command.



Stage 3
CBJ BroadwayJob1
params

Inputs
flowName : string
uid : string
schedule : string
Immediate

- Immediate
- Interval
- Date/Time
- Cron

wj: ~/workspace/project
Output

Job Types

User job

A Java function defined under a specific LU type or in Shared Objects.

The execution of the job is always tied to a specific LU type. If defined in Shared Objects, it can be executed across all LU types.

User job can be executed using the **startJob** command or **DbCommand Actor** (running the startJob command).

```
@desc("Test User Job")
@type(UserJob)
@out(name = "", type = String.class, desc = "")
public static String userJob1 (@desc("") int inputParameter) throws Exception {
    ...
  }
}
```

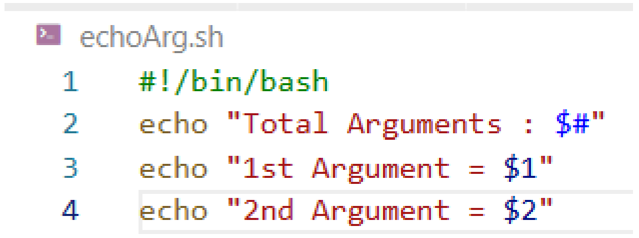
Job Types

Process job

Runs a script or executable stored on the Fabric server. It can be executed using the **DbCommand Actor** or the **startJob** command

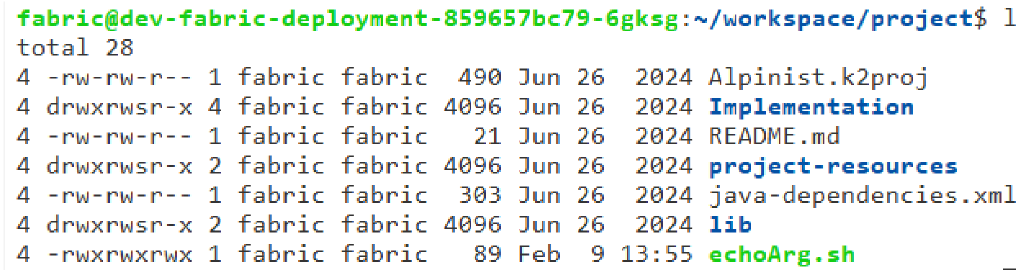
Example execution command:

startjob process NAME='/opt/apps/fabric/workspace/project/echoArg.sh' UID='processJobtest' ARGS='{"0":"ARG 1 value","1":"ARG 2 value"}';



echoArg.sh

```
#!/bin/bash
echo "Total Arguments : $#"
echo "1st Argument = $1"
echo "2nd Argument = $2"
```



```
fabric@dev-fabric-deployment-859657bc79-6gksg:~/workspace/project$ l
total 28
4 -rw-rw-r-- 1 fabric fabric  490 Jun 26  2024 Alpinist.k2proj
4 drwxrwsr-x 4 fabric fabric 4096 Jun 26  2024 Implementation
4 -rw-rw-r-- 1 fabric fabric   21 Jun 26  2024 README.md
4 drwxrwsr-x 2 fabric fabric 4096 Jun 26  2024 project-resources
4 -rw-rw-r-- 1 fabric fabric  303 Jun 26  2024 java-dependencies.xml
4 drwxrwsr-x 2 fabric fabric 4096 Jun 26  2024 lib
4 -rwxrwxrwx 1 fabric fabric   89 Feb  9 13:55 echoArg.sh
```

Job Types

Interface Listener Job

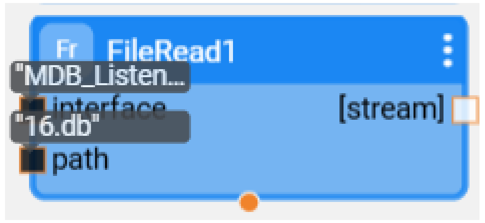
A listener to a storage (e.g., local file system) implemented by creating an **Interface** and a Broadway flow with the **InterfaceListener Broadway Actor**.
Once the flow is executed (only needs to be executed once), a job will start with the following parameters:

1. **Type** = INTERFACE_LISTENER
2. **Name** = The Broadway flow attached to the InterfaceListener
3. **UID** = The Interface name

**The interface will react only to newly added files*

Note:

If using the FileRead actor in the attached Broadway flow, set the parameters as external to receive inputs from the Interface Listener



Job Types

Batch (migrate) job

Created when executing a batch command. A job will start with the following parameters:

1. **Type** = BATCH_JOB
2. **UID** = The batch command

Common Jobs

See more info in Alpinist's Common course.

COMMONARE_TABLE_SYNC – executes Common table population

COMMONAREA_TABLE_REPLICATE – executed Common data's replicate request from another node in the cluster

CDC Job

Will be covered in the CDC course.

Job Commands

How to Start a Job?

- **Using the startjob Command**
  STARTJOB <JOBTYPE> NAME='<name>' [UID='<uid>'] [AFFINITY='<affinity>'] [ARGS='<args>'] [EXEC_INTERVAL='<execInterval>’];
  - Example: startjob user_job name='Customer.userJob1' args='{"loopsIterationCount":"2000"}' UID='uid_test';
- **Using a Broadway Flow**
  - **DbCommand Actor** – Select “fabric” interface and use the startJob command.
  - **BroadwayJob Actor** – Used specifically for Broadway jobs.
- **Using Java:**
  - fabric().fetch("startjob user_job name='Customer.userJob1' args='{\"loopsIterationCount\":\"2000\"}' UID='uid_test’”);
  - Example Java Method:

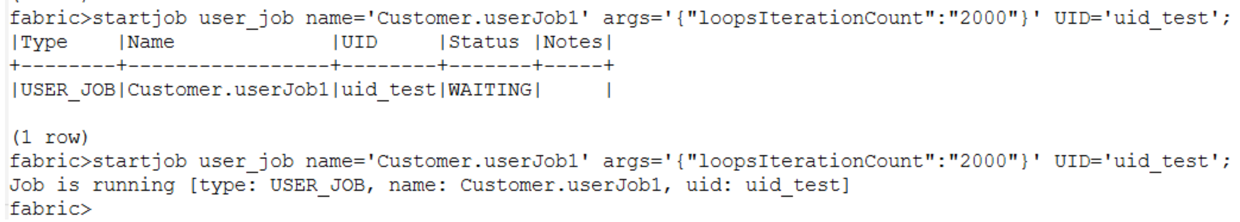
```
@desc("")
@out(name = "functionResult", type = String.class, desc = "")
public static Db.Row executeUserJob1(@desc("") String param1) throws Exception {
   Db.Row results = fabric().fetch("startjob user_job name='Customer.userJob1’ args='{\"loopsIterationCount\":\"2000\"}’
UID='uid_test'").firstRow();
   return results;
}
```

Job Commands

Job Parameters

STARTJOB <JOBTYPE> NAME='<name>' [UID='<uid>'] [AFFINITY='<affinity>'] [ARGS='<args>'] [EXEC_INTERVAL='<execInterval>'];

- **Job Type** – Specifies the type of job (e.g., BROADWAY, PROCESS, USER_JOB, etc.).
- **UID** – A unique identifier for the job. If not provided, Fabric will generate a new UID for each run.
  Using a UID ensures that the job cannot be started multiple times simultaneously. Only one instance of a job with the same UID is allowed.



```
fabric>startjob user_job name='Customer.userJob1' args='{"loopsIterationCount":"2000"}' UID='uid_test';
|Type    |Name             |UID     |Status |Notes|
+--------+-----------------+--------+-------+-----+
|USER_JOB|Customer.userJob1|uid_test|WAITING|     |

(1 row)
fabric>startjob user_job name='Customer.userJob1' args='{"loopsIterationCount":"2000"}' UID='uid_test';
Job is running [type: USER_JOB, name: Customer.userJob1, uid: uid_test]
fabric>
```

- **Affinity** – Restricts the job to run on specific nodes (more info in the Jobs Config section). Default value: {"Affinity":["ANY"]} – can run on any node.
- **Args** – A JSON string containing custom parameters for the job.
  Example: args='{"first_param":"first_value","second_param":"second_value"}'

Job Commands

Job Parameters

- **EXEC_INTERVAL** – Defines the job's execution schedule. If not provided, the job runs immediately and only once.
    - Timestamp: yyyy-MM-dd HH:mm:ss → Schedules a one-time execution.
    - Time Interval: HH:MM:SS → Runs the job at regular intervals.
    - Cron Expression: Uses the crontab format for complex scheduling.
      Example: 23 0-20/2 03 12 2 → Runs at minute 23, every 2nd hour from 0 to 20, on the 3rd day of the month, and on Tuesday in December.

Job Commands

How to stop a Job?

Stop job

- To stop a running job, use the STOPJOB command:
  - **Stop all jobs of a specific type and name:**
    STOPJOB <JOBTYPE> NAME='<name>'
    Stop all jobs matching the specified name and type.
  - **Stop a specific job using its UID:**
    STOPJOB <JOBTYPE> NAME='<name>' UID='<uid>'
    Stop only the job that matches the given UID.
- For **BW flows**, use the stopJob actor.

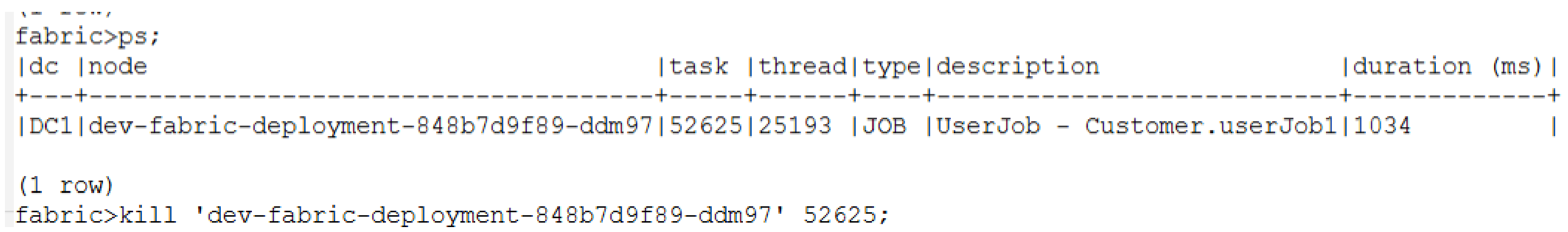
Job Commands

How to stop a Job?

Kill job

If a job does not respond to the **stopJob** command (e.g., a long-running SELECT on a source database that does not respond to an abort request) , you can forcefully terminate its thread using the **kill** command:

kill <node_id> <task_id>;



```
fabric>ps;
|dc |node                                  |task |thread|type|description                |duration (ms)|
+---+--------------------------------------+-----+------+----+---------------------------+-------------+
|DC1|dev-fabric-deployment-848b7d9f89-ddm97|52625|25193 |JOB |UserJob - Customer.userJob1|1034         |

(1 row)
fabric>kill 'dev-fabric-deployment-848b7d9f89-ddm97' 52625;
```

Note:

The ps command returns all threads on the current node. Use ps all to list threads running across the entire cluster.

Job Commands

How to Restart a Job?

Restart Job

Stop and start a non-archived job.

- RESTARTJOB <JOBTYPE> NAME='<name>'
  Restarts all matching Jobs with this name and type.
- RESTARTJOB <JOBTYPE> NAME='<name>' UID='<uid>'
  Restarts a specific Job matching an UID.

Resume Job

Start a job that is already marked with archived=true

- RESUMEJOB <JOBTYPE> NAME='<name>' UID='<uid>'
  Resumes a specific matching Job. This command applies only to an existing Job.

Job Commands

How to Update a Job?

To modify job parameters, use the **UPDATEJOB** command. You can update the following:

- **Affinity** – Restrict the job to specific nodes.
- **Arguments** – Update built-in and custom parameters.
- **Execution Interval** – Modify job scheduling.
- **RESET_END_TIME** (for recurring jobs only):
  - TRUE – Triggers the next execution immediately.
  - FALSE – Keeps the original schedule.
  - *To convert a cron job into a one-time job, set EXEC_INTERVAL=''.*

Command Syntax:

```
UPDATEJOB <jobType> NAME='<name>' [UID='<uid>']
[AFFINITY='<affinity>'] [ARGS='<args>'] [EXEC_INTERVAL='<execInterval>']
[RESET_END_TIME=true/false]
```

Jobs Monitoring

How to Monitor Jobs?

Use the JOBSTATUS command to monitor job execution and status.

Command Variants:

1. **Retrieve active or past jobs:** JOBSTATUS [x days ago]
   - If no days are provided, returns all active (non-archived) jobs.
   - If days are specified, returns the status of jobs executed in the past *X* days, including archived jobs.
2. **Retrieve jobs by type:** JOBSTATUS <JOBTYPE>
   - Returns the status of all jobs matching the specified type.
2. **Retrieve a specific job by type, name, and UID:** JOBSTATUS <JOBTYPE> '<NAME>' WITH UID='<UID>'
   - Returns the status of the specified job.

Jobs Monitoring

How to Monitor Jobs?

JOBSTATUS Output:

- **Type** – Job type (e.g., user_job, process, Broadway).
- **Name** – Job name.
- **UID** – Unique identifier of the job.
- **Status** – Current job state.
- **Creation Time** – When the STARTJOB command was last executed.
- **Start Time** – Start time of the last run.
- **End Time** – End time of the last run.
- **Affinity** – Node restriction, if any.
- **Is Archived** – Automatically set to True if the job reaches a terminated, failed, or processed state.
- **Next Run:**
  - IN_PROCESS – "Already running."
  - WAITING – "Ready to be executed."
  - SCHEDULED – Timestamp of the next scheduled execution for recurring jobs.
- **Ownership Candidates Num** – Number of nodes eligible to execute the job based on affinity settings.
- **Notes** – Last recorded error message if the job failed.
- **Node** – The node currently handling the job.
- **Tries** – Number of retry attempts in case of failure.

Session Scope

When a job is executed, Fabric automatically includes session scope variables in the job's arguments. Any SET command executed within the session is added to the session_scope parameter of the job.

This means that when a Session Global is set and a job is run within the same session, the job will inherit these global values and operate accordingly.

For example, executing the following commands:

- *SET sync off;*
- *SET auto_mdb_scope = true;*
- *SET k2_ws.ALPINIST = 10;*

Will result in the following job arguments:

*{"session_scope":"{\"scope\":{\"EXECUTION_ID\":\"e03e1922-ccbe-42d1-94ce-9470412e578d\",\"**k2_ws.ALPINIST**\":\"**10**\",\"**SYNC**\":\"**off**\",\"**AUTO_MDB_SCOPE**\":\"**true**\",\"LOG_ID\":\"9404000000001753\"}}"}*

* **execution_id** – Used internally by Fabric along with **log_id** for log tracing.

Jobs Mechanism

K2_JOBS table

Each STARTJOB command adds a record to the **k2System.k2_jobs** table:

- **TYPE:** Specifies the job type (BROADWAY_JOB, USER_JOB, COMMONAREA_TABLE_SYNC etc.
- **NAME:** The job name, including the associated LUT.
- **UID :** The unique identifier for the job
- **AFFINITY:** The node or DC (IP address) where the job can run on.
- **ARCHIVED:**
  - Automatically set to True when the job reaches a **terminated, failed, or processed** state.
  - If not specified, the default **TTL (Time-to-Live)** is used, set in config.ini (K2JOB_ARCHIVING_TIME_HOUR=720, equivalent to 30 days).
  - TTL applies at the row level when the job is archived.

Jobs Mechanism

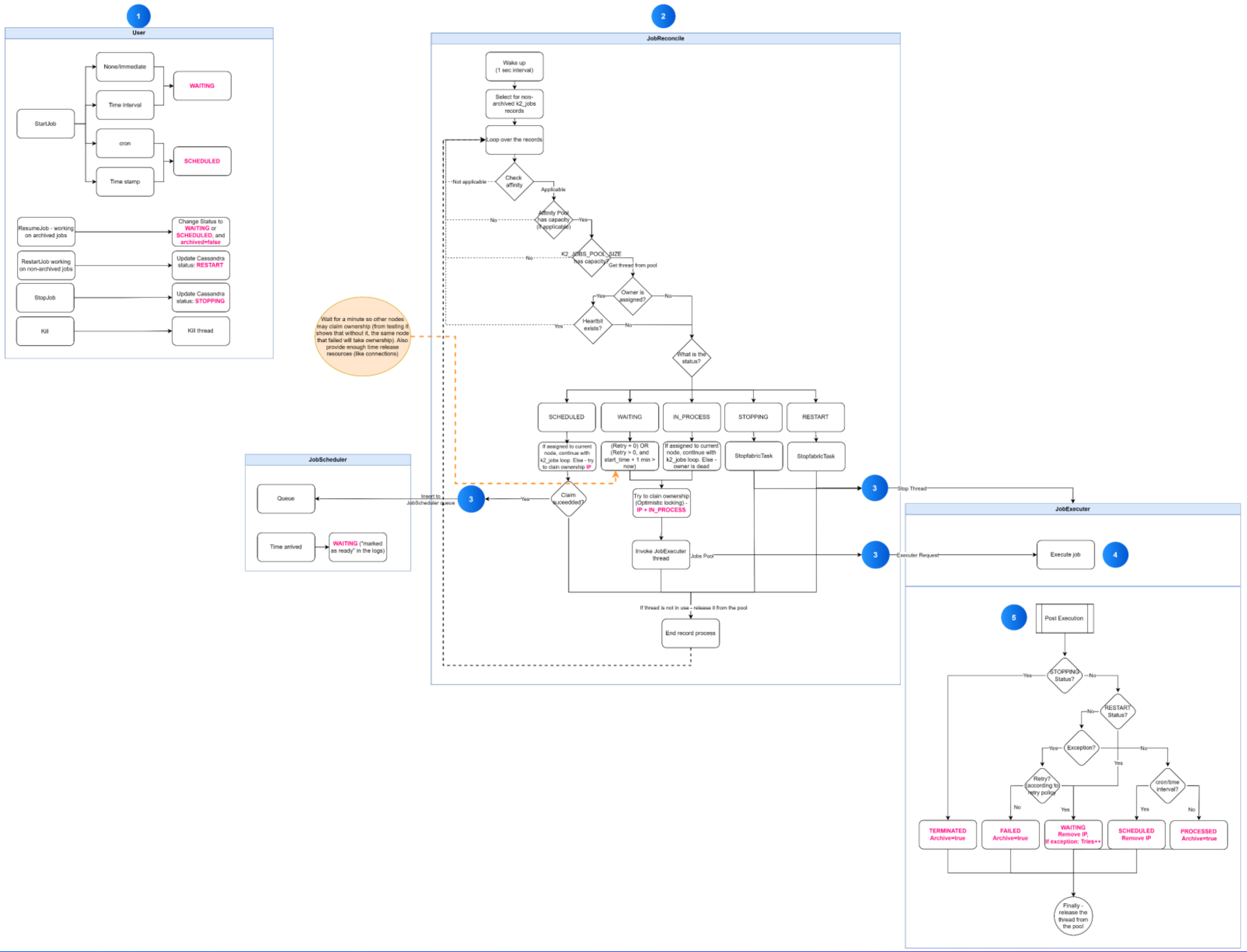
K2_JOBS table

- **ARGUMENTS:** Parameters passed to the job, including both session parameters and custom parameters.
- **CREATION_TIME:** Timestamp of the last startJob execution.
- **EXECUTION_INTERVAL:** Defined only for recurring jobs.
- **STATUS:** Current status of the job.
- **Last Run Statistics:**
  - **START_TIME:** Timestamp when the job last started.
  - **END_TIME:** Timestamp when the job last completed.
  - **ERROR_MSG:** Error message from the last run, if applicable. *(Not cleared even if a later run is successful?)*
  - **OUTPUT:** Stores job output if configured.
- **TRIES:** Number of retries (in case of failures).
- **WORKER_ID:** The node handling the job.

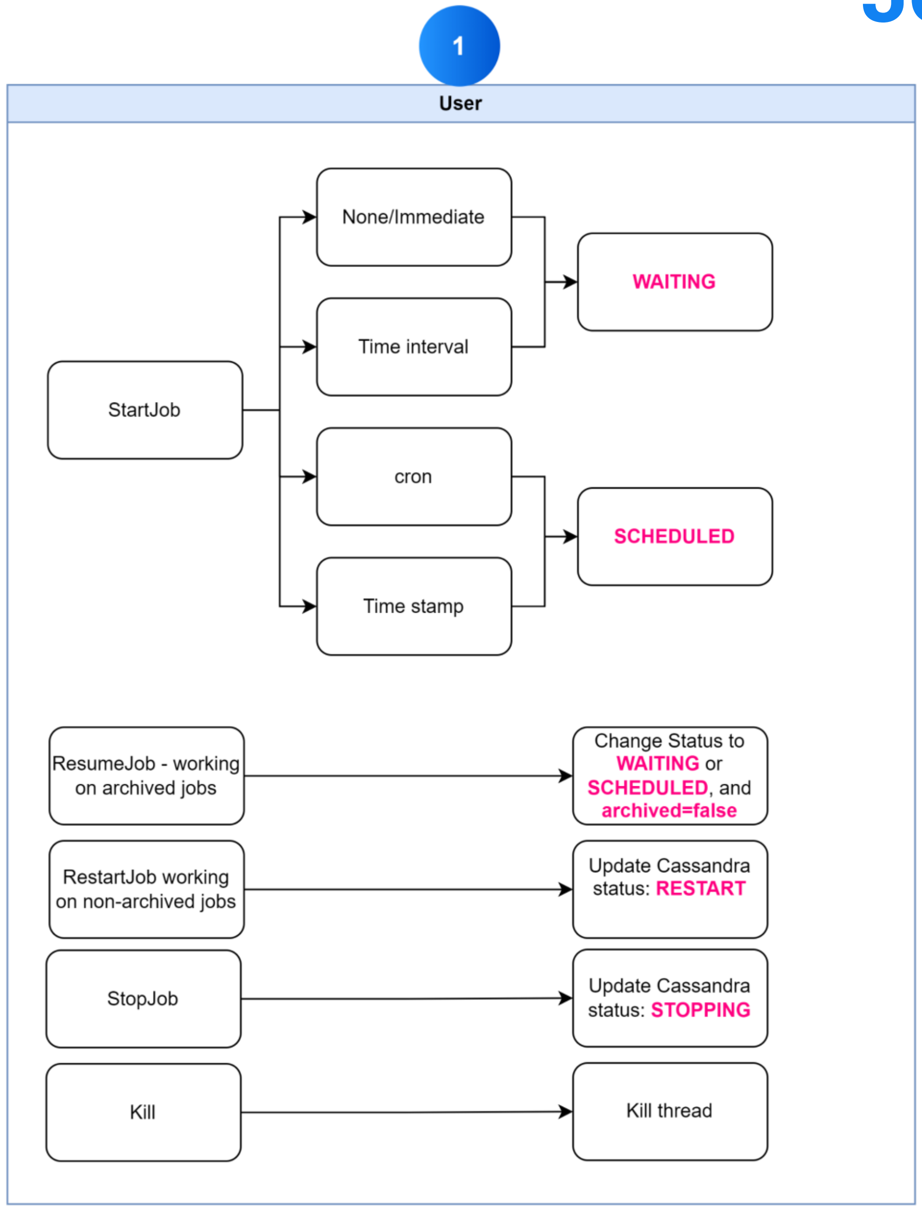
Job Execution with UID

If a job is started with a **UID** that already exists in **k2_jobs** (in **PROCESSED** status), the same record is **updated** instead of creating a new one.

Jobs Mechanism


1
User
StartJob
None/Immediate
Time interval
cron
Time stamp
WAITING
SCHEDULED
ResumeJob - working on archived jobs
Change Status to WAITING or SCHEDULED, and archived=false
RestartJob working on non-archived jobs
Update Cassandra status: RESTART
StopJob
Update Cassandra status: STOPPING
Kill
Kill thread
Wait for a minute so other nodes may claim ownership (from testing it shows that without it, the same node that failed will take ownership). Also provide enough time release resources (like connections)
2
JobReconcile
Wake up (1 sec interval)
Select for non-archived k2_jobs records
Loop over the records
Check affinity
Not applicable
Applicable
Affinity Pool has capacity (if applicable)
No
Yes
K2_JOBS_POOL_SIZE has capacity?
No
Get thread from pool
Owner is assigned?
Yes
No
Heartbit exists?
Yes
No
What is the status?
SCHEDULED
WAITING
IN_PROCESS
STOPPING
RESTART
If assigned to current node, continue with k2_jobs loop. Else - try to claim ownership IP
(Retry = 0) OR (Retry > 0, and start_time + 1 min > now)
If assigned to current node, continue with k2_jobs loop. Else - owner is dead
StopfabricTask
StopfabricTask
Claim suceedded?
Yes
3
Insert to JobScheduler queue
Try to clain ownership (Optimistic locking) - IP + IN_PROCESS
Invoke JobExecuter thread
Jobs Pool
If thread is not in use - release it from the pool
End record process
JobScheduler
Queue
Time arrived
WAITING ("marked as ready" in the logs)
3
Stop Thread
3
Executer Request
JobExecuter
Execute job
4
5
Post Execution
STOPPING Status?
Yes
No
RESTART Status?
No
Exception?
Yes
No
Yes
Retry? (according to retry policy)
No
Yes
cron/time interval?
Yes
No
TERMINATED Archive=true
FAILED Archive=true
WAITING Remove IP, If exception: Tries++
SCHEDULED Remove IP
PROCESSED Archive=true
Finally - release the thread from the pool

Jobs Mechanism


1
User
StartJob
None/Immediate
Time interval
cron
Time stamp
WAITING
SCHEDULED
ResumeJob - working on archived jobs
Change Status to WAITING or SCHEDULED, and archived=false
RestartJob working on non-archived jobs
Update Cassandra status: RESTART
StopJob
Update Cassandra status: STOPPING
Kill
Kill thread

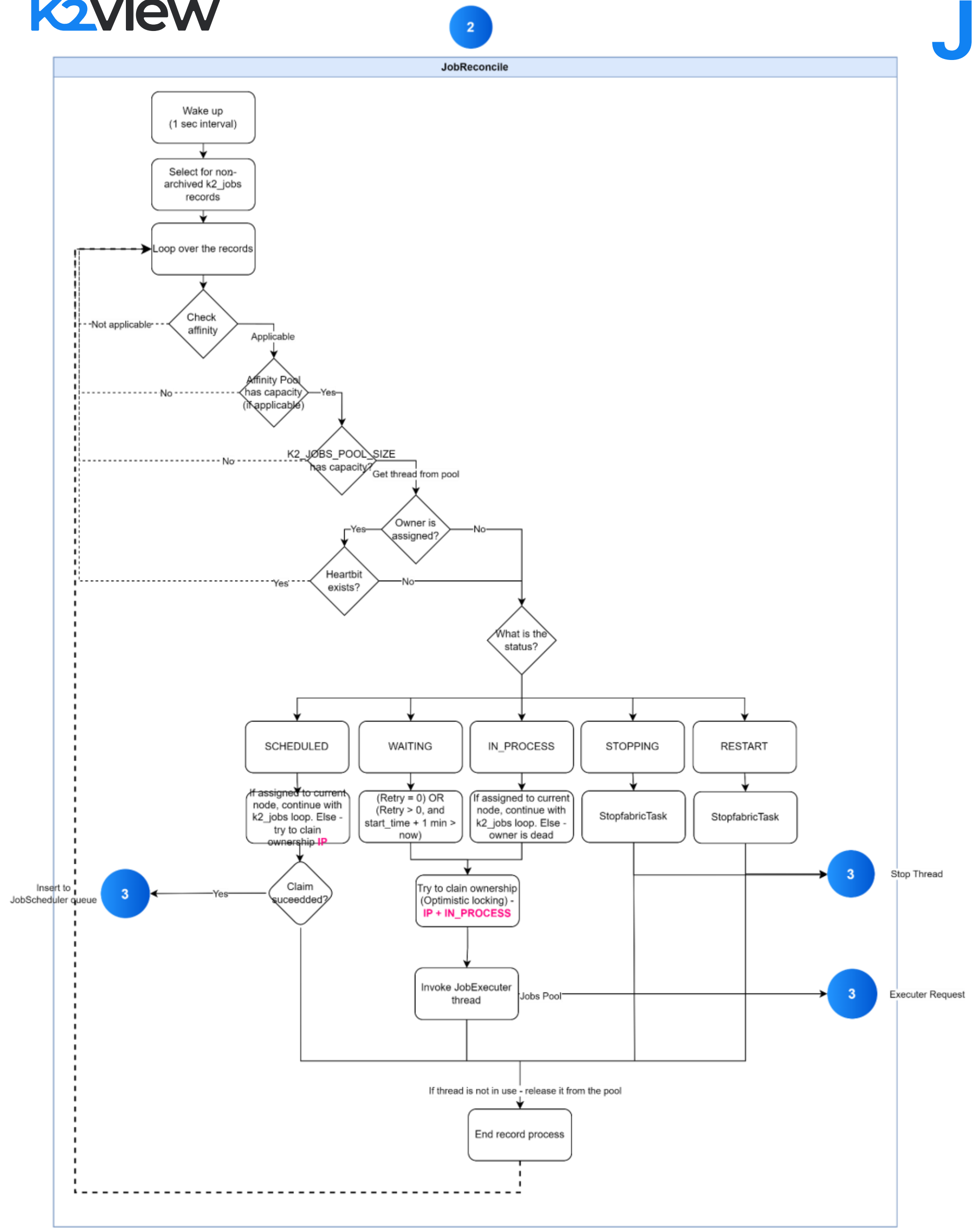
Job Lifecycle - User Action

A user starts a new job using the startJob command.

The startJob command creates a record in k2System.k2_jobs within the system database, setting the status based on the Schedule parameter:

- WAITING → If Schedule is Empty, Immediate, or Time Interval.
- SCHEDULED → If Schedule is Cron or Timestamp.

Jobs Mechanism

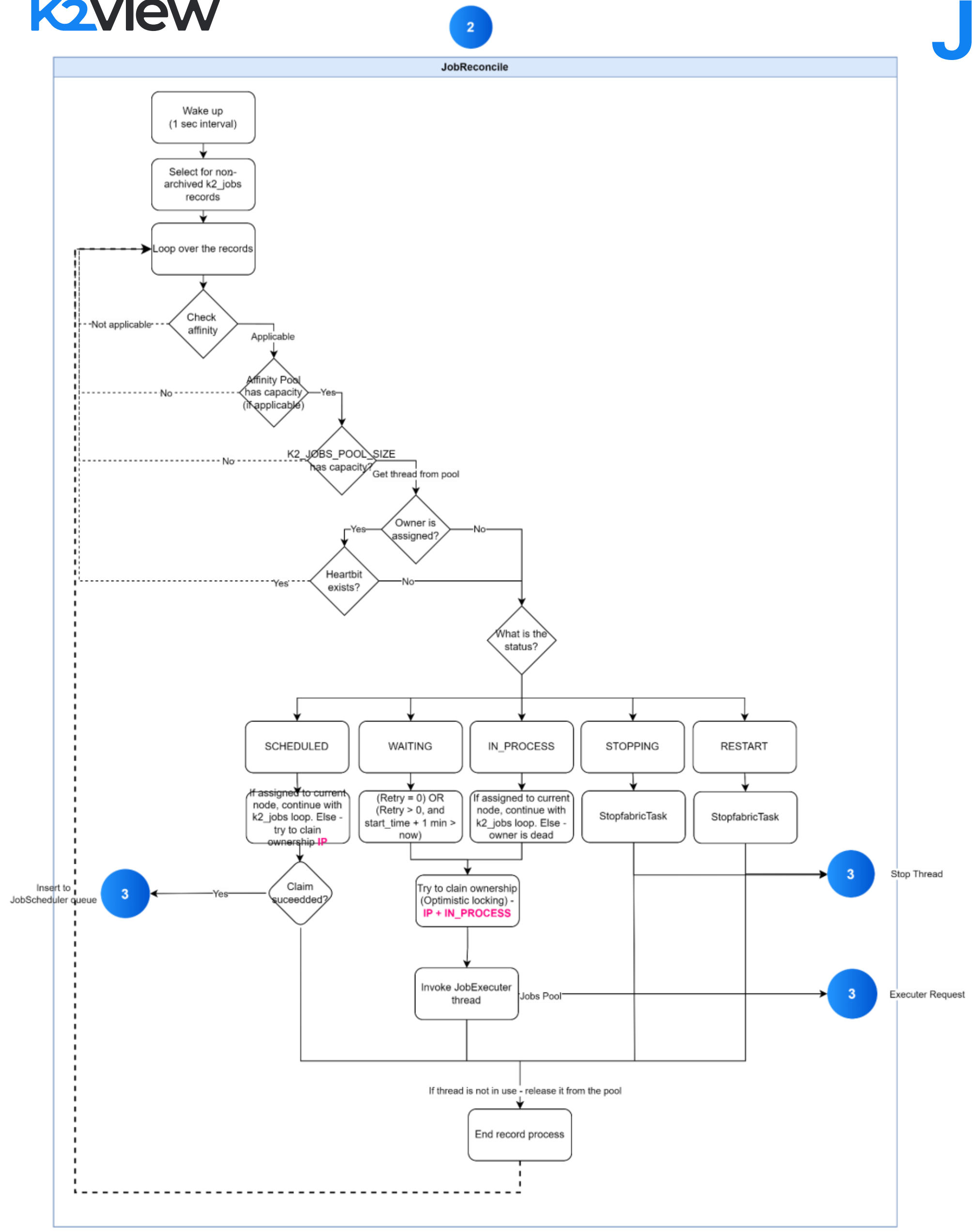

2
JobReconcile
Wake up (1 sec interval)
Select for non-archived k2_jobs records
Loop over the records
Check affinity
Not applicable
Applicable
Affinity Pool has capacity (if applicable)
No
Yes
K2_JOBS_POOL_SIZE has capacity?
No
Get thread from pool
Owner is assigned?
Yes
No
Heartbit exists?
Yes
No
What is the status?
SCHEDULED
WAITING
IN_PROCESS
STOPPING
RESTART
If assigned to current node, continue with k2_jobs loop. Else - try to clain ownership IP
(Retry = 0) OR (Retry > 0, and start_time + 1 min > now)
If assigned to current node, continue with k2_jobs loop. Else - owner is dead
StopfabricTask
StopfabricTask
Claim suceedded?
Yes
3
Insert to JobScheduler queue
Try to clain ownership (Optimistic locking) - IP + IN_PROCESS
Invoke JobExecuter thread
Jobs Pool
3
Stop Thread
3
Executer Request
If thread is not in use - release it from the pool
End record process

Job Lifecycle - JobReconcile

JobReconcile is a Fabric component that manages job execution and termination. It runs on every Fabric node and follows the process outlined in the flow diagram on the left.

- When a job is in the **SCHEDULED** status and claimed by a node, JobReconcile requests the JobScheduler to monitor it and transition it to **WAITING** when the execution time arrives.
- When a job is ready to run, JobReconcile starts a new thread of **JobExecuter**, which handles the job execution.

Jobs Mechanism



2
JobReconcile
Wake up (1 sec interval)
Select for non-archived k2_jobs records
Loop over the records
Check affinity
Not applicable
Applicable
Affinity Pool has capacity (if applicable)
No
Yes
K2_JOBS_POOL_SIZE has capacity?
No
Get thread from pool
Owner is assigned?
Yes
No
Heartbit exists?
Yes
No
What is the status?
SCHEDULED
WAITING
IN_PROCESS
STOPPING
RESTART
If assigned to current node, continue with k2_jobs loop. Else - try to claim ownership IP
(Retry = 0) OR (Retry > 0, and start_time + 1 min > now)
If assigned to current node, continue with k2_jobs loop. Else - owner is dead
StopfabricTask
StopfabricTask
Claim suceedded?
Yes
3
Insert to JobScheduler queue
Try to claim ownership (Optimistic locking) - IP + IN_PROCESS
3
Stop Thread
Invoke JobExecuter thread
Jobs Pool
3
Executer Request
If thread is not in use - release it from the pool
End record process

Job Lifecycle - JobReconcile

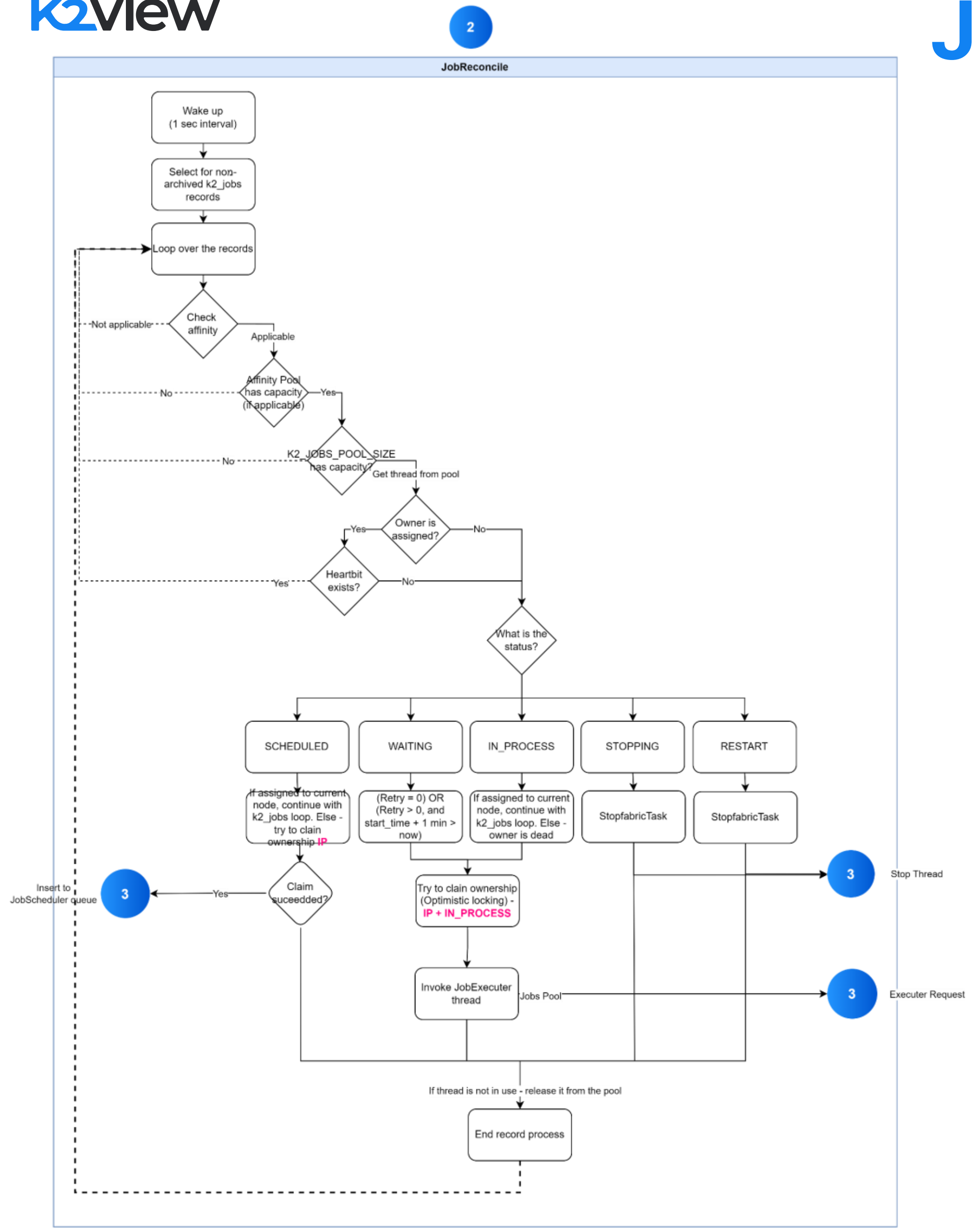
Claiming Job Ownership

A job becomes available for ownership by a node in the following cases:

1. **WAITING** – The job is ready for execution.
   - If **tries > 0** (meaning the job has failed), the node must wait **1 minute** before reclaiming it. This delay allows resources to be released and gives other nodes a chance to claim the job. *Tests have shown that without this 1-minute delay, the failing node would immediately reclaim it again.
   - Logs when job is executed:
     INFO 2025-01-27 21:55:21,616 [LID570400000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - 'USER_JOB.Customer.userJob6.uid_test6' **is now IN_PROCESS by dev-fabric-deployment-848b7d9f89-ddm97**

Jobs Mechanism

Job Lifecycle - JobReconcile

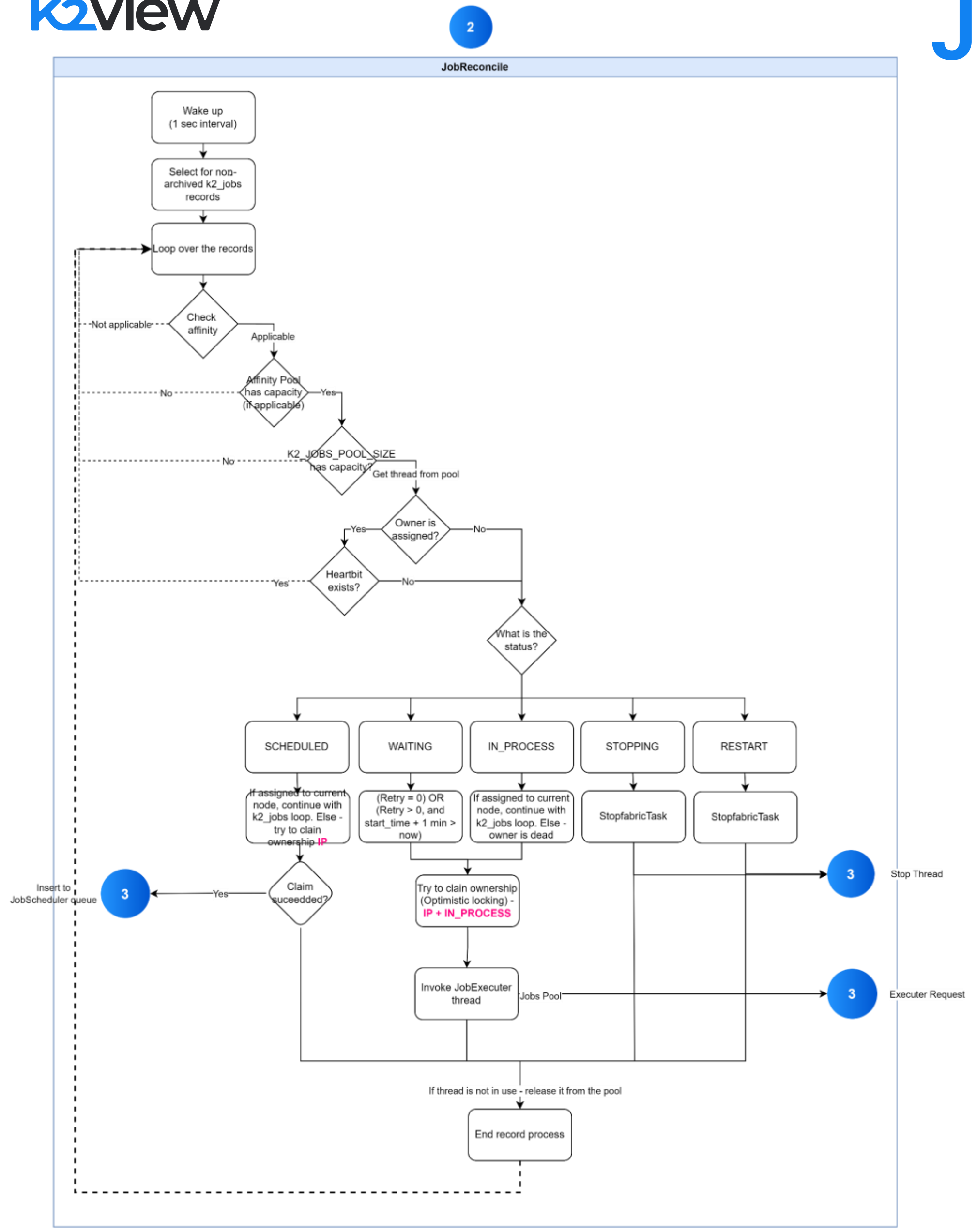

2
JobReconcile
Wake up
(1 sec interval)
Select for non-archived k2_jobs records
Loop over the records
Check affinity
Not applicable
Applicable
Affinity Pool has capacity (if applicable)
No
Yes
K2_JOBS_POOL_SIZE has capacity?
No
Get thread from pool
Owner is assigned?
Yes
No
Heartbit exists?
Yes
No
What is the status?
SCHEDULED
WAITING
IN_PROCESS
STOPPING
RESTART
If assigned to current node, continue with k2_jobs loop. Else - try to clain ownership IP
(Retry = 0) OR (Retry > 0, and start_time + 1 min > now)
If assigned to current node, continue with k2_jobs loop. Else - owner is dead
StopfabricTask
StopfabricTask
Claim suceedded?
Yes
3
Insert to JobScheduler queue
Try to clain ownership (Optimistic locking) - IP + IN_PROCESS
Invoke JobExecuter thread
Jobs Pool
3
Stop Thread
3
Executer Request
If thread is not in use - release it from the pool
End record process

2. **SCHEDULED** – The job is set to run at a specific time but requires an active node to execute it.

3. **IN_PROCESS** – The job was running, but the assigned node is no longer available.

To claim ownership, JobReconcile updates its IP in the worker_id column of the k2_jobs table. Fabric ensures that only one node can claim ownership at a time by using a lightweight transaction, preventing conflicts.

Jobs Mechanism


2
JobReconcile
Wake up (1 sec interval)
Select for non-archived k2_jobs records
Loop over the records
Check affinity
Not applicable
Applicable
Affinity Pool has capacity (if applicable)
No
Yes
K2_JOBS_POOL_SIZE has capacity?
No
Get thread from pool
Owner is assigned?
Yes
No
Heartbit exists?
Yes
No
What is the status?
SCHEDULED
WAITING
IN_PROCESS
STOPPING
RESTART
If assigned to current node, continue with k2_jobs loop. Else - try to clain ownership IP
(Retry = 0) OR (Retry > 0, and start_time + 1 min > now)
If assigned to current node, continue with k2_jobs loop. Else - owner is dead
StopfabricTask
StopfabricTask
Claim suceedded?
Yes
3
Insert to JobScheduler queue
Try to clain ownership (Optimistic locking) - IP + IN_PROCESS
3
Stop Thread
Invoke JobExecuter thread
Jobs Pool
3
Executer Request
If thread is not in use - release it from the pool
End record process

Job Lifecycle - JobReconcile

Example Log Entry for Successful Ownership Claim:

INFO 2025-01-27 20:56:31,218 [LID5704000000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - dev-fabric-deployment-848b7d9f89-ddm97 **successfully claimed ownership of** cron job 'COMMONAREA_TABLE_SYNC.COMMONAREA_TABLE_SYNC.COMMONAREA_TABLE_SYNC_ref_crm.payment'

Note: When a node restarts, its worker ID is not deleted from the k2_jobs table, giving it precedence over other nodes for executing its assigned jobs.

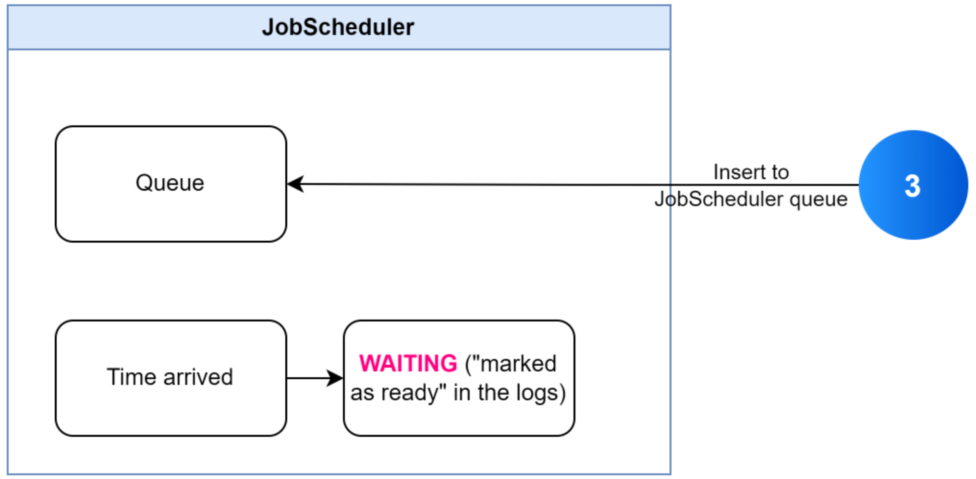
Jobs pool

Each node maintains a pool of active jobs running on it.

- When a job thread starts, it is added to the jobs pool and removed once the thread terminates.
- The maximum number of concurrent jobs per node is defined in config.ini: K2JOBS_POOL_SIZE=25
- If the pool reaches its limit, the node cannot claim new jobs until a slot becomes available.

Jobs Mechanism

Job Lifecycle – JobScheduler



JobScheduler
Queue
Insert to JobScheduler queue
3
Time arrived
WAITING ("marked as ready" in the logs)

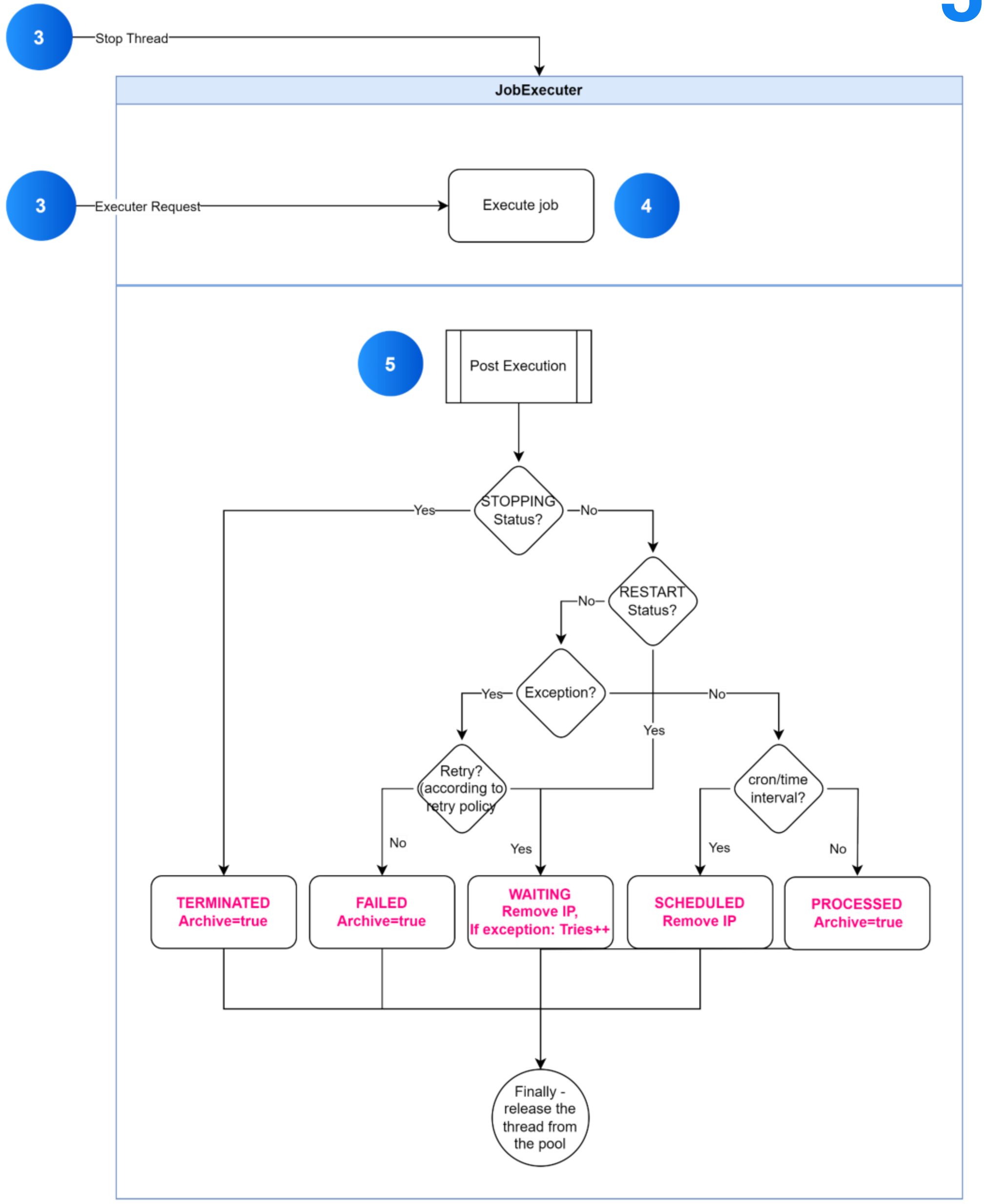
The **JobScheduler** is a Fabric component responsible for tracking a node's scheduled jobs and moving them to **WAITING** when their execution time arrives.

It maintains a **queue of scheduled jobs**, received from **JobReconcile**, for monitoring.

When a job reaches its scheduled time, the following log entry is recorded:

INFO 2025-01-27 21:55:20,611 [LID5704000000000337] [InternalQueue-ref_crm.payment] c.k.c.j.JobsScheduler - cron job 'USER_JOB.Customer.userJob6.uid_test6' **marked as ready**

Jobs Mechanism



Job Lifecycle – jobExecuter

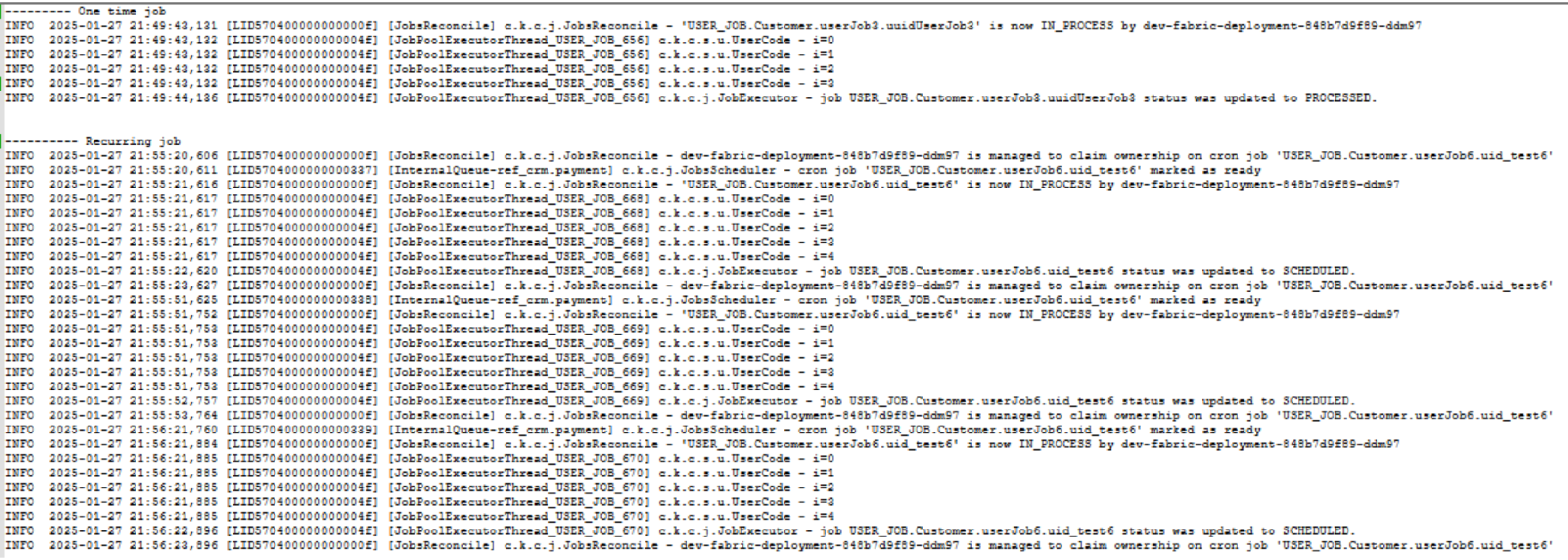
The JobExecuter is a thread initiated by JobReconcile, responsible for executing jobs and updating their status upon completion.

Once a job completes, the Post Execution method is triggered to handle status updates:

- **STOPPING** – If the job was stopped by a user (stopJob command), the status is updated to **TERMINATED**.
- **RESTART** – If the job was restarted (restartJob command) or a deployment occurred, the status is updated to **WAITING**, and the Owner is removed.
- **Exception** Raised – If the job was terminated by a kill command or encountered an exception:
  - If retries are allowed, the status is set to **WAITING**.
  - If no retries remain, the status is set to **FAILED**.
- Otherwise, the status, IP, and archived flag are updated according to the workflow to the left.

Jobs Mechanism

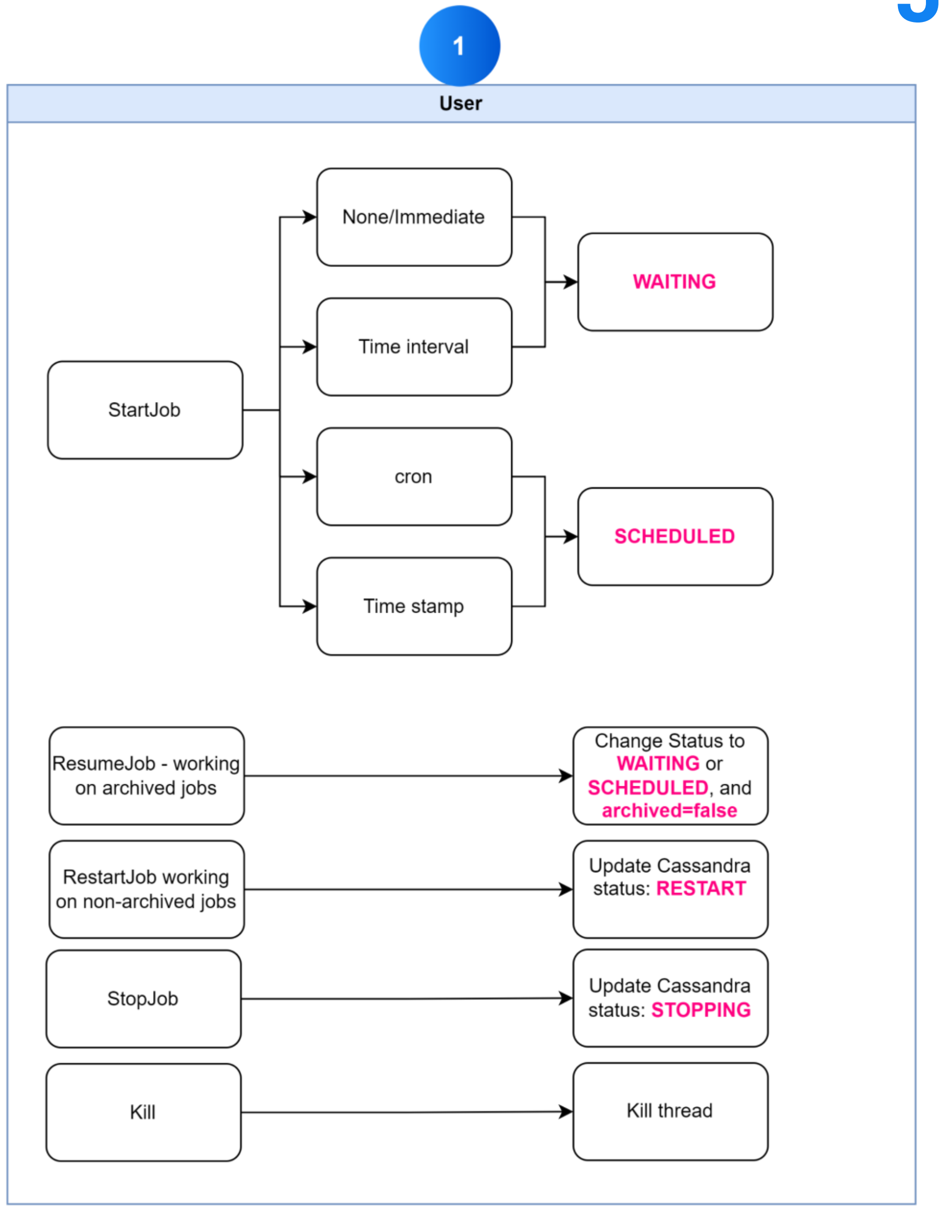
Job Lifecycle - Logs



```
--------- One time job
INFO  2025-01-27 21:49:43,131 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - 'USER_JOB.Customer.userJob3.uuidUserJob3' is now IN_PROCESS by dev-fabric-deployment-848b7d9f89-ddm97
INFO  2025-01-27 21:49:43,132 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_656] c.k.c.s.u.UserCode - i=0
INFO  2025-01-27 21:49:43,132 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_656] c.k.c.s.u.UserCode - i=1
INFO  2025-01-27 21:49:43,132 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_656] c.k.c.s.u.UserCode - i=2
INFO  2025-01-27 21:49:43,132 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_656] c.k.c.s.u.UserCode - i=3
INFO  2025-01-27 21:49:44,136 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_656] c.k.c.j.JobExecutor - job USER_JOB.Customer.userJob3.uuidUserJob3 status was updated to PROCESSED.


---------- Recurring job
INFO  2025-01-27 21:55:20,606 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - dev-fabric-deployment-848b7d9f89-ddm97 is managed to claim ownership on cron job 'USER_JOB.Customer.userJob6.uid_test6'
INFO  2025-01-27 21:55:20,611 [LID57040000000337] [InternalQueue-ref_crm.payment] c.k.c.j.JobsScheduler - cron job 'USER_JOB.Customer.userJob6.uid_test6' marked as ready
INFO  2025-01-27 21:55:21,616 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - 'USER_JOB.Customer.userJob6.uid_test6' is now IN_PROCESS by dev-fabric-deployment-848b7d9f89-ddm97
INFO  2025-01-27 21:55:21,617 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.s.u.UserCode - i=0
INFO  2025-01-27 21:55:21,617 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.s.u.UserCode - i=1
INFO  2025-01-27 21:55:21,617 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.s.u.UserCode - i=2
INFO  2025-01-27 21:55:21,617 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.s.u.UserCode - i=3
INFO  2025-01-27 21:55:21,617 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.s.u.UserCode - i=4
INFO  2025-01-27 21:55:22,620 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_668] c.k.c.j.JobExecutor - job USER_JOB.Customer.userJob6.uid_test6 status was updated to SCHEDULED.
INFO  2025-01-27 21:55:23,627 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - dev-fabric-deployment-848b7d9f89-ddm97 is managed to claim ownership on cron job 'USER_JOB.Customer.userJob6.uid_test6'
INFO  2025-01-27 21:55:51,625 [LID57040000000338] [InternalQueue-ref_crm.payment] c.k.c.j.JobsScheduler - cron job 'USER_JOB.Customer.userJob6.uid_test6' marked as ready
INFO  2025-01-27 21:55:51,752 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - 'USER_JOB.Customer.userJob6.uid_test6' is now IN_PROCESS by dev-fabric-deployment-848b7d9f89-ddm97
INFO  2025-01-27 21:55:51,753 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.s.u.UserCode - i=0
INFO  2025-01-27 21:55:51,753 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.s.u.UserCode - i=1
INFO  2025-01-27 21:55:51,753 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.s.u.UserCode - i=2
INFO  2025-01-27 21:55:51,753 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.s.u.UserCode - i=3
INFO  2025-01-27 21:55:51,753 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.s.u.UserCode - i=4
INFO  2025-01-27 21:55:52,757 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_669] c.k.c.j.JobExecutor - job USER_JOB.Customer.userJob6.uid_test6 status was updated to SCHEDULED.
INFO  2025-01-27 21:55:53,764 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - dev-fabric-deployment-848b7d9f89-ddm97 is managed to claim ownership on cron job 'USER_JOB.Customer.userJob6.uid_test6'
INFO  2025-01-27 21:56:21,760 [LID57040000000339] [InternalQueue-ref_crm.payment] c.k.c.j.JobsScheduler - cron job 'USER_JOB.Customer.userJob6.uid_test6' marked as ready
INFO  2025-01-27 21:56:21,884 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - 'USER_JOB.Customer.userJob6.uid_test6' is now IN_PROCESS by dev-fabric-deployment-848b7d9f89-ddm97
INFO  2025-01-27 21:56:21,885 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.s.u.UserCode - i=0
INFO  2025-01-27 21:56:21,885 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.s.u.UserCode - i=1
INFO  2025-01-27 21:56:21,885 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.s.u.UserCode - i=2
INFO  2025-01-27 21:56:21,885 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.s.u.UserCode - i=3
INFO  2025-01-27 21:56:21,885 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.s.u.UserCode - i=4
INFO  2025-01-27 21:56:22,896 [LID57040000000004f] [JobPoolExecutorThread_USER_JOB_670] c.k.c.j.JobExecutor - job USER_JOB.Customer.userJob6.uid_test6 status was updated to SCHEDULED.
INFO  2025-01-27 21:56:23,896 [LID57040000000000f] [JobsReconcile] c.k.c.j.JobsReconcile - dev-fabric-deployment-848b7d9f89-ddm97 is managed to claim ownership on cron job 'USER_JOB.Customer.userJob6.uid_test6'
```

Jobs Mechanism


1
User
StartJob
None/Immediate
Time interval
cron
Time stamp
WAITING
SCHEDULED
ResumeJob - working on archived jobs
Change Status to WAITING or SCHEDULED, and archived=false
RestartJob working on non-archived jobs
Update Cassandra status: RESTART
StopJob
Update Cassandra status: STOPPING
Kill
Kill thread

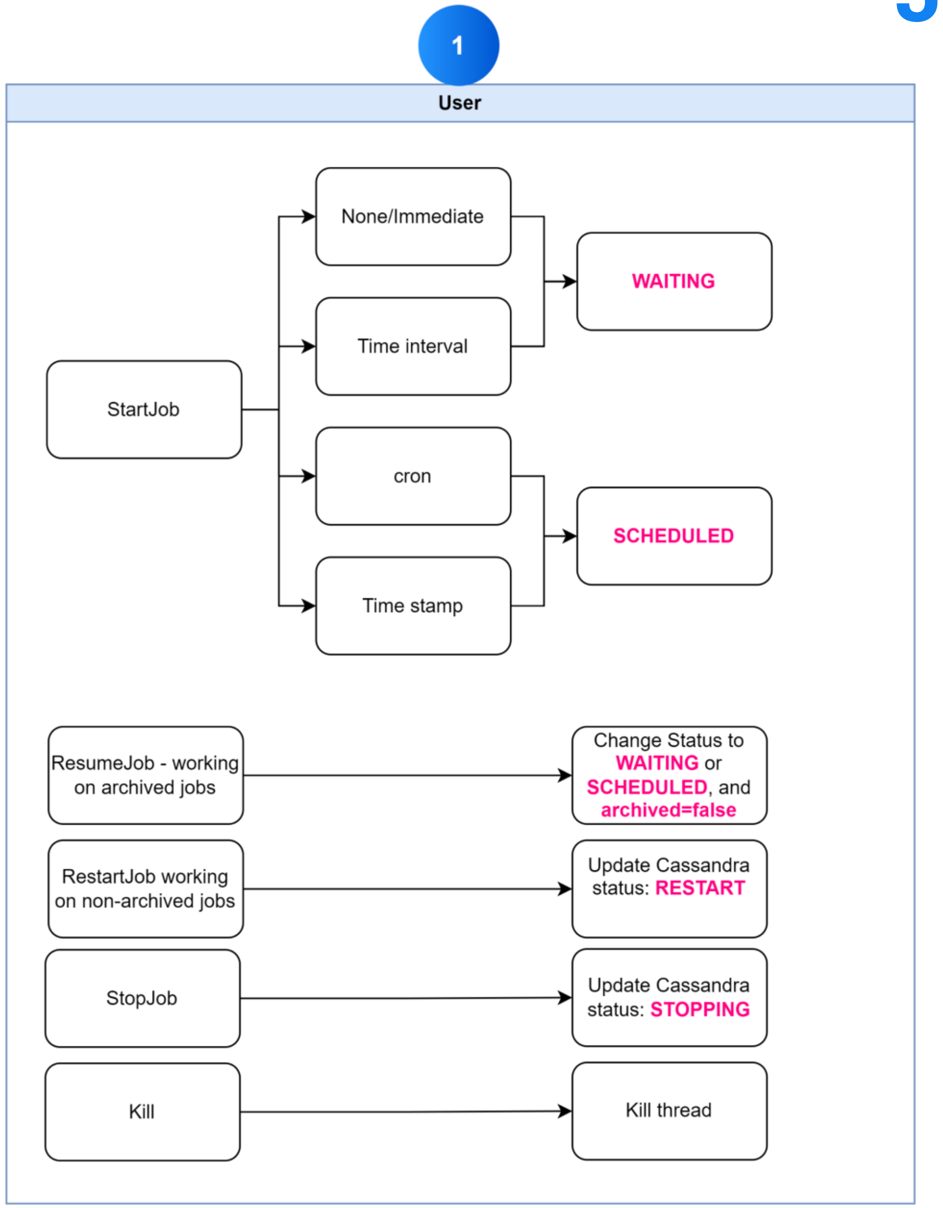
Managing Job Execution

Stopping a job

Executing the stopJob command updates the job's record to **STOPPING** status.

- **JobReconcile** detects the **STOPPING** status while scanning non-archived records and terminates the job's thread.
- During **post-execution**, **JobExecuter** updates the job's status to **TERMINATED** and sets **archived = true**.

Jobs Mechanism


1
User
StartJob
None/Immediate
Time interval
cron
Time stamp
WAITING
SCHEDULED
ResumeJob - working on archived jobs
Change Status to WAITING or SCHEDULED, and archived=false
RestartJob working on non-archived jobs
Update Cassandra status: RESTART
StopJob
Update Cassandra status: STOPPING
Kill
Kill thread

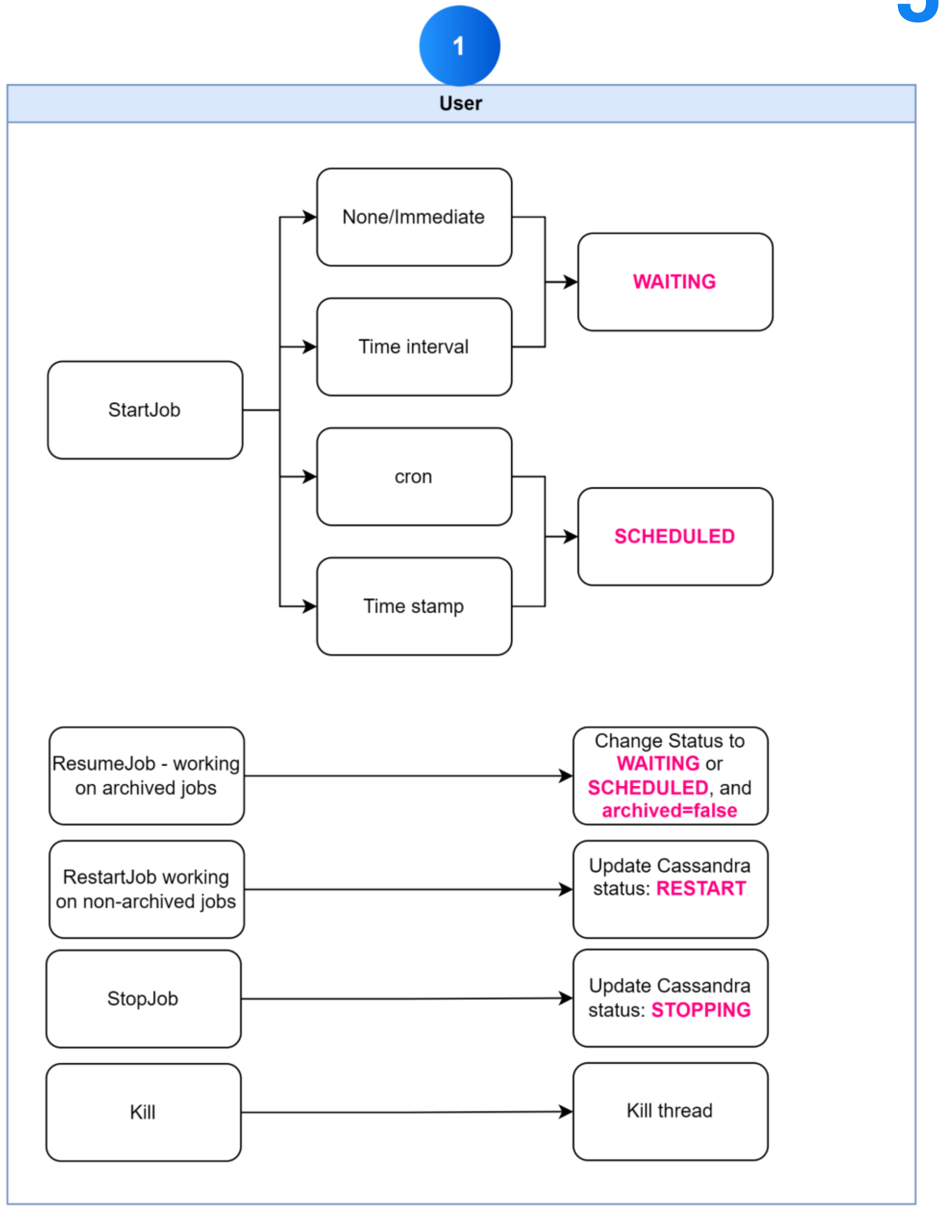
Managing Job Execution

Killing a job's thread

Fabric's **ps** and **kill** commands can be used to terminate a job's thread if the stopJob command fails to stop it (e.g., when the job does not handle the abort process correctly).

- In such cases, the thread is forcefully terminated.
- The post_execution method detects the error message and updates the job status accordingly:
  - **WAITING** – If the restart count has not reached the threshold.
  - **FAILED** – If the restart count limit has been exceeded.
- The thread is then closed and removed from the job pool.

Jobs Mechanism


1
User
StartJob
None/Immediate
Time interval
cron
Time stamp
WAITING
SCHEDULED
ResumeJob - working on archived jobs
Change Status to WAITING or SCHEDULED, and archived=false
RestartJob working on non-archived jobs
Update Cassandra status: RESTART
StopJob
Update Cassandra status: STOPPING
Kill
Kill thread

Managing Job Execution

Restarting a job

Executing the restartJob command updates the job's record to **RESTART** status.

- **JobReconcile** detects the **RESTART** status while scanning non-archived records, stops the job's thread, and updates the status to **WAITING**.
- A node will then claim ownership and execute the job.

Resuming a job

Executing the resumeJob command updates the job's record to **WAITING** or **SCHEDULED** (depending on its schedule) and sets **archived = false**. A **JobReconcile** instance will detect the job and claim ownership to execute it.

Job Implementation

Aborting a Job

Use the built-in function to actively handle the **stopJob** command invoked by the user:

```
if (isAborted()) {
        ...Add custom logic as required...
        throw new InterruptedException();
}
```

Note:

- When using BW flows, the Broadway mechanism automatically checks for abort signals between each stage.
- Explicit handling is typically required when subscribing to Kafka without using the **Subscribe** Broadway actor.

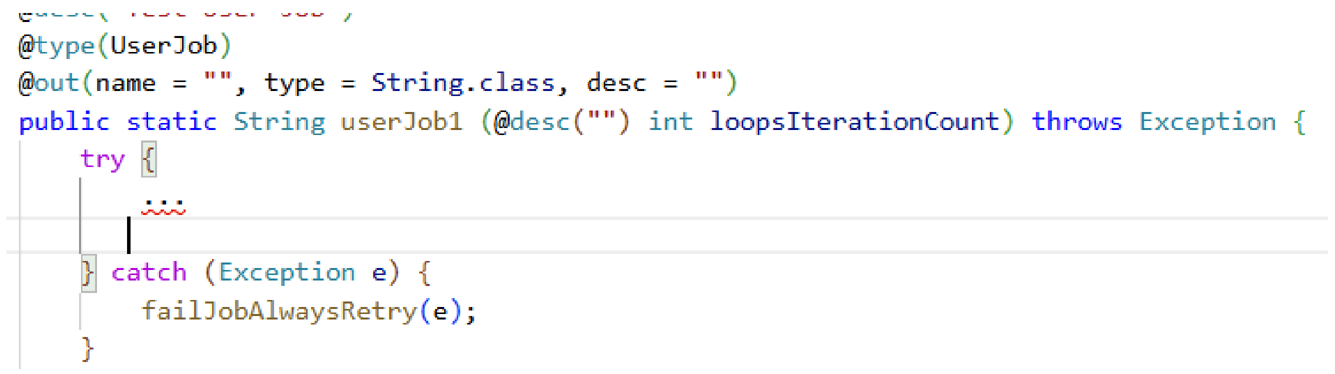
Job Implementation

Job retries

By default, Fabric attempts to execute a failing job 10 times before marking it as FAILED.

To override the retry mechanism, use:

- Java Code:
  - **failJobRetryUntilMax**(Throwable e) – Default behavior (retries until the max limit is reached).
  - **failJobAlwaysRetry**(Throwable e) – Forces the job to retry indefinitely, ignoring the retry limit.
  - **failJobNoRetry**(Throwable e) – Stops job execution immediately and prevents any retries.

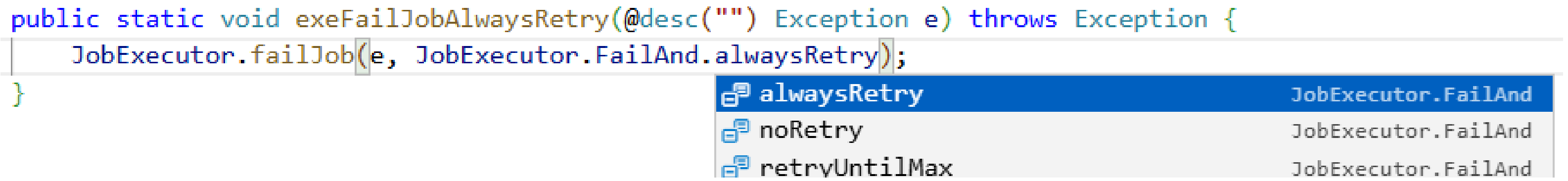


```java
@type(UserJob)
@out(name = "", type = String.class, desc = "")
public static String userJob1 (@desc("") int loopsIterationCount) throws Exception {
    try {
        ...
        
    } catch (Exception e) {
        failJobAlwaysRetry(e);
    }
```

Job Implementation

- Broadway (BW) Flow:
  - Use ErrorHandler to catch exceptions and trigger a flow.
  - The flow calls a LU function to execute the following:

import com.k2view.cdbms.jobs.JobExecutor;



```
public static void exeFailJobAlwaysRetry(@desc("") Exception e) throws Exception {
    JobExecutor.failJob(e, JobExecutor.FailAnd.alwaysRetry);
}
```

```
alwaysRetry        JobExecutor.FailAnd
noRetry            JobExecutor.FailAnd
retryUntilMax      JobExecutor.FailAnd
```

Note: The e parameter contains the exception details, which are:

- Logged in the system log file.
- Stored in the k2_jobs.error_msg column.
- Displayed when running the JobStatus command.

Job Implementation

Job output

When a job returns a result, the output is stored in the k2_jobs.**output** column.

Note: In Broadway (BW), the output includes all parameters defined as **External**.

Example Output:[{"result":"Loop executed 500 times","result2":222}]

Heartbeat

Fabric provides multiple recovery mechanisms to ensure job execution continuity if a node fails. A **heartbeat** mechanism monitors each Fabric node, allowing jobs to be reassigned when necessary.

Each node updates its heartbeat in **k2system.nodes** with the following behavior:

- Every node updates its status in Cassandra every **FABRIC_HEARTBEAT_INTERVAL_MS** milliseconds (default: **5 seconds**).
- A node is considered **inactive** if it misses **FABRIC_HEARTBEAT_MISS** updates (default: **12 misses**).
- If a node stops updating its heartbeat, other nodes assume it is **down** and can claim its jobs. The failing node will also terminate all its active job threads.

Note:

- Any job with affinity set exclusively to a failed node will not run and must be restarted manually.
- If a node restarts shortly before a scheduled job's execution, it takes precedence over other nodes for running that job. This is achieved by retaining the **worker_id** in the **k2_jobs** table before the restart.

Jobs & Project Deployment

- The **deploy.flow** can be used to start jobs automatically.
- Deploying a project restarts all running jobs associated with the deployed LUT, ensuring they run with the updated code.
- Recurring jobs that are not running at the time of deployment are not restarted.
- **Broadway and User jobs** restart automatically after deployment (or a Fabric restart), while **Parsers do not**.

Jobs Actors

JobWait

The JobWait actor waits for a specific job to complete or until a timeout occurs.

- **Mandatory Input Parameters:**
    - Job type
    - Job name
    - UID (The UID can be assigned during the startJob command and used here.)
- **Output:**
- Parameters from the k2_jobs record

Jobs Actors

StopJob

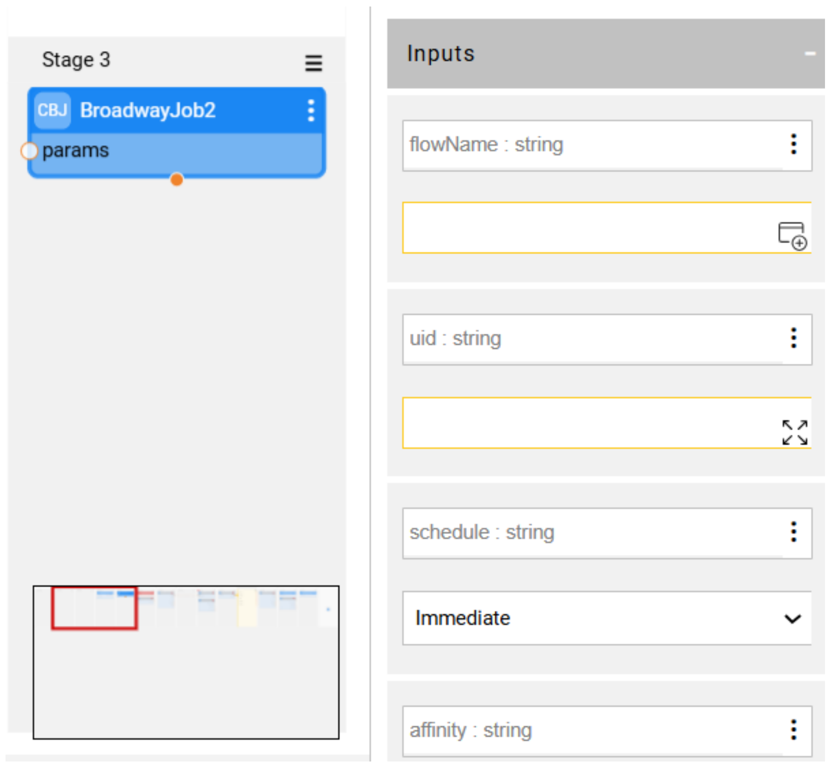
The StopJob actor stops a running job and waits until it completes, or a defined timeout is reached.

- **Mandatory Input Parameters:**
  - Job type
  - Job name
  - UID (The UID can be assigned during the startJob command and used here.)
- **Output:**
  - Job status

Jobs Actors

BroadwayJob

BroadwayJob actor provides the ability to trigger a **Fabric Job** that will in turn execute another Broadway flow once or multiple times depending upon the configuration of the job.



Inputs

flowName : string

uid : string

schedule : string

Immediate

affinity : string

Jobs Configuration

Config.ini

JOBS section:.

- **K2JOBS_POOL_SIZE=25**, defines the size of the thread pool for processing Fabric Jobs.
- **K2JOB_ARCHIVING_TIME_HOUR=720**, defines the time when to delete archived Job row in the **k2_jobs table**. Default is 720 hours (30 days).
- **CLAIM_EXCEPTIONAL_INTERVAL_SEC=60,** defines the wait interval before claiming a job with limited affinity, once the recommended pool size has been reached for this affinity (see next slide).

.

Jobs Configuration

Node.id

The **node.id** file, located in the **config** folder, is used to assign logical names to nodes. These logical names can be used to set **affinity** for specific jobs, ensuring that only nodes with the matching affinity can execute those jobs.

Affinity-Based Job Execution

For example, adding "HANDLE_MSG:5" to the node.id file of **Node 1** and **Node 2** allows these two nodes to execute up to **5 instances** of jobs with the **HANDLE_MSG** affinity.

- Nodes **without** HANDLE_MSG in node.id or with HANDLE_MSG:0 **cannot** execute these jobs.
- If more than **10 jobs** start simultaneously, each node will execute **5 jobs**, while the remaining jobs will stay in **WAITING** status until a slot becomes available.

.

Jobs Configuration

Node.id

ANY as affinity

The ANY option is **enabled by default** for all nodes and applies only to jobs **without a specific affinity**.

Configuring ANY Affinity in node.id:

1. **Exclude a Node:** Set ANY:0 to prevent a node from executing jobs without affinity.
2. **Limit Job Execution:** Set ANY:<value> to define the **maximum number of jobs** a node can run concurrently.
3. **Default Behavior:** If ANY is **not** specified in node.id, the node will automatically execute jobs without a specific affinity.

Jobs Configuration

Node.id

Affinity range (Fabric 6.4.2 and later):

Fabric allows defining an **affinity range** in node.id, e.g.,HANDLE_MSG:**5 10**

- **5** → Recommended maximum number of concurrent jobs.
- **10** → Absolute maximum number of concurrent jobs.

Defining a range can be used to cover of a dead node. How It Works:

- Nodes will **prioritize** staying within the **recommended limit** to allow other nodes to take jobs.
- If a node has reached its **recommended limit** but still tries to claim a new job, it must wait for a specific time before taking on additional jobs.

⏳ **Claim Delay:**

- This waiting time is controlled by the CLAIM_EXCEPTIONAL_INTERVAL_SEC parameter in **config.ini** (default: 60 seconds).
- During this time, other nodes with available slots get the first opportunity to claim jobs.

Jobs Configuration

Node.id – Affinity Range

Handling Excess Jobs

If a node exceeds its **recommended pool size**, Fabric will:

1. **Stop & Release** extra jobs running beyond the recommended limit.
2. **Allow Other Nodes** with available slots to claim the stopped jobs.

To avoid immediate job restarts on the same node, a **random delay** is applied before restarting the job. This delay is controlled by:

- MIN_GIVE_UP_EXCEPTIONAL_MINUTES
- MAX_GIVE_UP_EXCEPTIONAL_MINUTES

These parameters can be configured in **config.ini** under the jobs section.

Important Note: It is recommended to set the minimum affinity value to match the required number of job instances. Otherwise, jobs may continuously restart.

Jobs Configuration

Node.id – Affinity Range

Important Notes:

- Set the minimum affinity value to match the required number of job instances; otherwise, jobs may continuously restart.
- This is especially critical for jobs consuming from Kafka, as each restart triggers a rebalance, temporarily halting consumption and reducing the overall consumption rate.

JMX Stats

Under the Transactions section:

- **systemJobs** – Displays the number of currently running jobs (total 0 indicates the job is not running).
- **systemJobsExecution** – Shows the total number of times the job has been executed.