KIP-745: Connect API to restart connector and tasks

- Status
- Motivation
  - Example
  - How do the restart methods currently work?
- Scope
- Public Interfaces
  - REST API
  - Restart Method
    - Status Method
  - Metrics
- Proposed Changes
  - Connector and Task State
  - Standalone Runtime
  - Distributed Runtime
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives
  - Use REST API for Worker-to-Worker Communication
  - Synchronous REST API to Restart
  - Persist the Restart Requests the Status Topic

Status

Current state: Adopted

Discussion thread: *here*

Vote thread: *here*

JIRA: ⚠ Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

When a user runs a connector on Apache Kafka Connect, the framework starts one instance of the connector's `Connector` implementation class and one or more instances of the connector's `Task` implementation class. Any of these instances can experience an error. Generally if the `Connector` or `Task` inst ance throws an exception, the Connect framework will mark that instance as failed and expose it as FAILED through the Connect REST API.

Currently users must use the REST API status method and/or JMX metrics to monitor the health ("status") of each named connector's `Connector` and `Tas` instances. If any of those instances have failed, a user must issue a separate REST API call to manually restart each of the `Connector` and `Task` instances.

The Connect REST API should allow users to restart all failed `Connector` and Task instances using a single REST API call.

Example

Consider a connector named "my-connector" that is configured to run 3 tasks. When running nominally, the connector's status output might look like the following:

Sample Connector Status Response

```
GET /connectors/my-connector/status
200 OK
{
    "name": "my-connector",
    "connector": {
        "state": "RUNNING",
        "worker_id": "fakehost1:8083"
    },
    "tasks":
    [
        {
            "id": 0,
            "state": "RUNNING",
            "worker_id": "fakehost2:8083"
        },
        {
            "id": 1,
            "state": "RUNNING",
            "worker_id": "fakehost3:8083"
        },
        {
            "id": 2,
            "state": "RUNNING",
            "worker_id": "fakehost1:8083"
        }
    ]
}
```

Here we can see that "my-connector" has one `Connector` instance and three `Task` instances, and all have a RUNNING state. Additionally, we can see that the `Connector` instance and task 3 are running on "fakehost1", task 0 is running on "fakehost2", and task 1 is running on "fakehost1".

If tasks 1 and 2 were to experience non-retriable errors, the connector's status might look like the following:

Sample Connector Status Response With Some Failed Tasks

```
GET /connectors/my-connector/status
200 OK
{
    "name": "my-connector",
    "connector": {
        "state": "RUNNING",
        "worker_id": "fakehost1:8083"
    },
    "tasks":
    [
        {
            "id": 0,
            "state": "RUNNING",
            "worker_id": "fakehost2:8083"
        },
        {
            "id": 1,
            "state": "FAILED",
            "worker_id": "fakehost3:8083",
            "trace": "Unable to execute HTTP request: Connect to data.example.com:443 failed: connect timed out"
        },
        {
            "id": 2,
            "state": "FAILED",
            "worker_id": "fakehost1:8083",
            "trace": "Unable to execute HTTP request: Connect to data.example.com:443 failed: connect timed out"
        }
    ]
}
```

Here we can see that the `Connector` instance and the one of the three `Task` instances have a RUNNING state, but task 1 and 2 each have a FAILED state. If a user detects this and wants to restart the two failed tasks (task 1 and task 2), they have to manually request a restart of each of the failed tasks:

```
POST /connectors/my-connector/tasks/1/restart
```

and

```
POST /connectors/my-connector/tasks/2/restart
```

Some custom tooling can be built to extract the numbers of the failed tasks from the connector's status response, but Kafka Connect does not do any of this out of the box.

How do the restart methods currently work?

There are two restart methods in the Kafka Connect REST API:

- `POST /connectors/{connectorName}/restart` – restarts only the `Connector` instance for the named connector.
- `POST /connectors/{connectorName}/tasks/{taskNum}/restart` – restarts only the `Task` instance for the named connector and specified task number.

Most users expect that the first method restarts everything associated with the named connector. This is a sensible expectation, but it unfortunately does not at all align with the actual behavior described above.

A user can submit such restart requests to any worker. If the worker is assigned the corresponding `Connector` or `Task` instance, the worker will perform the restart. Otherwise, the worker is not assigned the instance and must ***forward*** the request to the worker that is assigned that instance. (If the worker receiving the request is not the leader, it actually forwards the request to the leader first, which then either restarts its assigned instance or forwards it to the worker assigned the instance.)

As the connector or task are restarted, changes in the status of `Connector` and `Task` instances are written to the internal status topic by the worker to which those instances are assigned. Every worker consumes this topic and therefore knows the current status of all `Connector` and `Task` instances for all named connectors. This is why ***any*** worker can respond to the `GET /connectors/{connectorName}/status` or `GET /connectors/{connectorName}/tasks/{taskNum}/status` methods.

Scope

The objective of this KIP is to modify the existing "connector restart" method in the Kafka Connect REST API to allows a user to issue **one request** to restart all or just the failed `Connector` and `Task` instances for a named connector. However, any changes should use optional query parameter that default to the existing behavior of just restarting just the `Connector` object when those optional query parameters are not supplied by clients.

Public Interfaces

REST API

Two existing methods of the Connect REST API will be changed: the connector restart method and the connector status method. No other REST API methods will be changed.

Restart Method

The existing method to request a restart of the `Connector` object for a named connector is as follows:

```
POST /connectors/{connectorName}/restart
```

This existing restart method in the Connect REST API will be changed to align with the natural expectation of many users: begin to restart a combination of the `Connector` and `Task` instances associated with the named connector. The user will specify which combination via two (2) new optional boolean query parameters that default to "false", called "includeTasks" and "onlyFailed":

```
POST /connectors/{connectorName}/restart?includeTasks=<true|false>&onlyFailed=<true|false>
```

where the behavior of these parameters is as follows:

- the "includeTasks" parameter specifies whether to restart the connector instance and task instances ("includeTasks=true") or just the connector instance ("includeTasks=false"), and defaults to "false".
- the "onlyFailed" parameter specifies whether to restart just the instances with a FAILED status ("onlyFailed=true") or all instances ("onlyFailed=false"), and defaults to "false".

The default value of these new query parameters is such that invocations without them result in the same old behavior of this REST API method: the `Connector` instance is restarted regardless of its current status, and no `Task` instances are restarted. The query parameters must be used to restart any other combination of the `Connector` and/or `Task` instances for the named connector. This approach satisfies the backward compatibility requirement.

The response of this method will be changed slightly, but will still be compatible with the old behavior (where "includeTasks=true" and "onlyFailed=false"). When these new query parameters are used with "includeTasks" and/or "onlyFailed" set to true, a successful response will be `202 ACCEPTED`, signaling that the request to restart some subset of the `Connector` and/or `Task` instances was accepted and will be asynchronously performed.

- `202 ACCEPTED` when the named connector exists and the server has successfully and durably recorded the *request* to stop and begin restarting at least one failed or running `Connector` object and `Task` instances (e.g., "includeTasks=true" or "onlyFailed=true"). A response body will be returned, and it is similar to the `GET /connector/{connectorName}/status` response except that the "state" field is set to RESTARTING for all instances that will eventually be restarted.
- `204 NO CONTENT` when the named connector exists and the server has successfully stopped and begun restarting only the `Connector` object (e.g., "includeTasks=false" and "onlyFailed=false"). No response body will be returned (to align with the existing behavior).
- `404 NOT FOUND` when the named connector does not exist.
- `409 CONFLICT` when a rebalance is needed, forthcoming, or underway while restarting any of the `Connector` and/or `Task` objects; the reason may mention that the Connect cluster's leader is not known, or that the worker assigned the Connector cannot be found.
- `500 Internal Server Error` when the request timed out (takes more than 90 seconds), which means the request could not be durably recorded, perhaps because the worker or cluster are shutting down or because the worker receiving the request has temporarily lost contact with the Kafka cluster.

Note that this method is asynchronous: a 202 ACCEPTED response indicates that the request to restart was accepted, and that all `Connector` and/or `Task` objects specified in the request will eventually be restarted. This closely aligns with the other asynchronous REST API methods (e.g., create connector, delete connector, etc.), and is better able to scale to connectors with large numbers of tasks.

Using our example above, we could use this expanded restart method to request the restart of all **failed** `Connector` and/or `Task` instances:

Example Usage of Restart Connector and Tasks

```
POST /connectors/my-connector/restart?includeTasks=true&onlyFailed=true
202 ACCEPTED
{
    "name": "my-connector",
    "connector": {
        "state": "RUNNING",
        "worker_id": "fakehost1:8083"
    },
    "tasks":
    [
        {
            "id": 0,
            "state": "RUNNING",
            "worker_id": "fakehost2:8083"
        },
        {
            "id": 1,
            "state": "RESTARTING",
            "worker_id": "fakehost3:8083"
        },
        {
            "id": 2,
            "state": "RESTARTING",
            "worker_id": "fakehost1:8083"
        }
    ]
}
```

The 202 ACCEPTED response signifies that the request to restart the instances has (at least) been accepted and enqueued for completion. This restart might take some time depending upon the state of the Connect cluster and the number of tasks in the connector, but it will eventually be performed in its entirety.

Also, the response signals that the `Connector` instance and task 0 were not going to be restarted. The request included the "onlyFailed=true" query parameter, and both the `Connector` instance and task 0 were RUNNING when this call was made. However, if either of these instances were to fail *before* their assigned workers process the restart request, they too would be restarted. Note that if a worker is restarted before it processes the restart request, it may skip its share of the restart request since the worker will restart all of its assigned `Connector` and `Task` instances as part of its normal startup process.

The state of each restarting instance will eventually transition to RUNNING once when assigned worker (re)starts that instance. The user can monitor this progress with subsequent calls to the `GET /connector/{connectorName}/status` method, though a user using the REST API to monitor the status of the connector may not observe all of these transitions between RESTARTING and RUNNING.

Status Method

The existing REST API method will also be changed slightly:

```
GET /connector/{connectorName}/status
```

The response of this method will be changed to include "RESTARTING" as a possible value for the "state" fields. This aligns with the `POST /connectors/{connectorName}/restart?includeTasks=true`.

Metrics

The following existing metrics will be modified:

MBean Name	Change	New Range of Values
kafka.connect:type=connector-metrics,name=status, connector=([-.\w]+)	Add "restarting" as a possible value for this metric.	one of: "unassigned", "running", "paused", "failed", "restarting" or "destroyed"
kafka.connect:type=connector-task-metrics, name=status,connector=([-.\w]+)	Add "restarting" as a possible value for this metric.	one of: "unassigned", "running", "paused", "failed", "restarting" or "destroyed"

Proposed Changes

The Connect distributed and standalone runtimes will be changed to support restarting a combination of the `Connector` and `Task` instances for a named connector, based upon the new query parameters. The restart requests will result in the state of each targeted `Connector` and `Task` instances being initially set to RESTARTING, and will reuse the worker's existing logic to stop and (asynchronously) start the `Connector` and `Task` instances. That existing logic does not wait for the instances to complete their startup, and it will transition this state to RUNNING when the instance is actually started.

Connector and Task State

A new RESTARTING state will be added to the `AbstractStatus.State` enumeration, and all corresponding code using this will be changed accordingly. Only the herders will explicitly set the RESTARTING state in the state store and in the metrics; the states of each instance will transition to RUNNING when the instance is restarted. The details of how this happens differs in the Standalone and Distributed runtimes.

Standalone Runtime

The standalone runtime runs all `Connector` and `Task` instances within the same VM, so it's straightforward to change the `StandaloneHerder` to respond to a restart connector and tasks by performing the following actions:

1. Get the current states of the `Connector` and `Task` instances and determine which the herder will target for restart.
2. If at least one instance is to be restarted:
    a. Stop and await all targeted instances.
    b. Set the RESTARTING state for these targeted instances.
    c. Restart all targeted instances, which will transition the states to RUNNING.
3. Build a `ConnectorStateInfo` result based upon the original status, with "state=RESTARTING" for all instances that were restarted.

The standalone runtime will ensure that each restart request is processed completely before the response is returned, even though the new API contract does not require this.

Distributed Runtime

The distributed runtime runs all `Connector` and `Task` instances across multiple VMs, and uses a combination of the config and status Kafka topics and rebalance logic to manage these instances. When a connector and tasks are restarted, the `DistributedHerder` will perform the following actions:

1. Write a new "restart request" command to the config topic.
2. Get the current states of the `Connector` and `Task` instances and determine which the herder will target for restart.
3. Set the RESTARTING state for all targeted instances.
4. Build and return a `ConnectorStateInfo` result based upon the current connector status, except with "state=RESTARTING" for all restarted instances that are targeted for restart.

The `202 ACCEPTED` response signifies that the "restart request" has been durably written to the config topic and all the workers in the Connect cluster will (eventually) see the restart request. If the worker reads the restart request as part of worker startup, it can ignore the restart request since the worker will subsequently attempt to start all of its assigned `Connector` and `Task` instances, effectively achieving the goal of restarting the instances assigned to that worker. If the worker reads the restart request *after* worker startup, then the `DistributedHerder` will enqueue the request to be processed within the herder's main thread loop. During this main thread loop, the herder will dequeue all pending restart requests and for each request use the current connector status and the herder's current assignments to determine which of *its* `Connector` and `Task` instances are to be restarted, and will then stop and restart them. Note that because this is done within the main thread loop, the herder will not concurrently process any assignment changes while it is executing the restart requests.

The "restart request" written to the config topic, which already is where the connector and task config records, task state change records, and session key records are written. This topic also make sense since all records related to restarts and configuration changes are totally ordered, and are all processed within the herder's main thread loop. The "restart request" records will not conflict with any other types of config records, will be compatible with the compacted topic, and will look like:

```
key: “restart-connector-<connectorName>”
value: {“include-tasks”: <true|false>, "only-failed": <true|false>}
```

Compatibility, Deprecation, and Migration Plan

The proposed API changes are entirely backward compatible. Restarting a named connector with the default query parameters results in always restarting only the `Connector` instance, which is the same behavior as in previous releases.

Rejected Alternatives

Use REST API for Worker-to-Worker Communication

When a worker receives a restart request via the REST API, it could determine which `Connector` and `Task` instances are to be restarted, and then issues a REST API restart request to each worker to signal the instances that worker should restart. However, this fan-out pattern had several disadvantages. Alth ough unlikely, it still is possible that the original restart request could time out if the delegated restart requests each take a long time. Second, the implementation would have been more complex to parallelize the delegated restart requests to each worker. Third, it is less reliable as network errors, rebalances, and other interruptions might result in only a subset of the targeted instances being restarted, especially when the number of workers is large.

On the other hand, the current approach is more reliable, since once the restart request is written to the config topic it will be eventually consumed by all workers. The current proposal also builds upon and reuses much more of the existing functionality in the worker, making the overall implementation more straightforward. There is also no chance for changing worker assignments to interfere with the restarts, since the current approach performs the restarts during the same herder thread loop that reacts to all rebalance changes. And, the new approach is more efficient, as some restart requests can be ignored if the worker will subsequently (re)start its assigned instances. For example, if a restart for a connector is requested but one of the worker is itself restarted (or joins the cluster), the worker as part of startup will start all of its assigned `Connector` and `Task` instances, making the restart unnecessary.

Synchronous REST API to Restart

Although somewhat related to the previous rejected alternative, it would still be possible to define the restart operation as synchronous and use either worker-to-worker communication or the config topic approach and return a response only when all instances have been stopped and restarted.

The current proposal makes the restart method asynchronous because making it synchronous has a few disadvantages. First, most of the other REST API methods that deal with stopping or starting connectors are asynchronous, because those operations can potentially be long-running. Second, this is exacerbated with connectors that have large numbers of tasks, or connector implementations that do not stop in an adequate time window. (Strictly speaking the latter is already addressed via other fixes, but even having the restart request be potentially-long running leads to potentially poor user experience.) Third, by combining this with the config topic based approach we can achieve much higher guarantees that the restart will be processed.

Persist the Restart Requests the Status Topic

As mentioned above, the proposal is to write the new "restart records" to the config topic. This makes a lot of sense, especially since it makes sense that the restart requests are totally ordered with respect to other configuration-related records in the config topic. Plus, there is precedence for other kinds of "non-config" records in the config topic.

However, it would also be possible to store the restart records in the status topic. Unfortunately, the StatusBackingStore interface does not define a listener mechanism, and adding that would require more effort and be more complex. Plus, it's not clear that persisting the restart requests in the status topic is any better than the config topic.