KIP-108: Create Topic Policy

Status

Current state: *Adopted*

Discussion thread: here

JIRA: KAFKA-4591

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

Motivation

A number of Admin APIs have been/will be introduced as part of KIP-4. A subset of these Admin APIs allow the user to create, update or delete cluster resources: create topics, delete topics, alter topics, alter ACLs and alter configs. Create and delete topics were added to Kafka in 0.10.0.0 while the others are still in progress. These APIs are a major step towards self-serve Kafka clusters where application developers can, for example, create topics without having to go through the admins/operations team and without access to ZooKeeper.

There is, however, no way to validate the user request in order to restrict the operation parameters. The operations team may want to enforce that the replication factor, min.insync.replicas and/or retention settings for a topic are within an allowable range for example. In this KIP, we limit ourselves to validation of the create topic request, but the approach can be used for other request types in the future.

Public Interfaces

A user can define a policy manager similar to the pluggable Authorizer by setting `create.topic.policy.class.name` in `server.properties` and implementing the the `CreateTopicPolicy` interface. The interface will live in the clients jar under the `org.apache.kafka.server.policy` package. It implements `Configurable` so that implementations can act on broker configurations and `AutoCloseable` so that resources can be released on shutdown.

```
package org.apache.kafka.server.policy;

/**
 * An interface for enforcing a policy on create topics requests.
 *
 * Common use cases are requiring that the replication factor, min.insync.replicas and/or retention settings
for a
 * topic are within an allowable range.
 *
 * If <code>create.topic.policy.class.name</code> is defined, Kafka will create an instance of the specified
class
 * using the default constructor and will then pass the broker configs to its <code>configure()</code> method.
During
 * broker shutdown, the <code>close()</code> method will be invoked so that resources can be released (if
necessary).
 */
public interface CreateTopicPolicy extends Configurable, AutoCloseable {

    /**
     * Class containing the create request parameters.
     */
    class RequestMetadata {
        private final String topic;
        private final Integer numPartitions;
        private final Short replicationFactor;
        private final Map<Integer, List<Integer>> replicasAssignments;
        private final Map<String, String> configs;

        /**
```

```
         * Create an instance of this class with the provided parameters.
         *
         * This constructor is public to make testing of <code>CreateTopicPolicy</code> implementations easier.
         *
         * @param topic the name of the topic to created.
         * @param numPartitions the number of partitions to create or null if replicasAssignments is set.
         * @param replicationFactor the replication factor for the topic or null if replicaAssignments is set.
         * @param replicasAssignments replica assignments or null if numPartitions and replicationFactor is
set. The
         *                             assignment is a map from partition id to replica (broker) ids.
         * @param configs topic configs for the topic to be created, not including broker defaults. Broker
configs are
         *                passed via the {@code configure()} method of the policy implementation.
         */
        public RequestMetadata(String topic, Integer numPartitions, Short replicationFactor,
                        Map<Integer, List<Integer>> replicasAssignments, Map<String, String> configs) {
            this.topic = topic;
            this.numPartitions = numPartitions;
            this.replicationFactor = replicationFactor;
            this.replicasAssignments = replicasAssignments == null ? null : Collections.unmodifiableMap
(replicasAssignments);
            this.configs = Collections.unmodifiableMap(configs);
        }

        /**
         * Return the name of the topic to create.
         */
        public String topic() {
            return topic;
        }

        /**
         * Return the number of partitions to create or null if replicaAssignments is not null.
         */
        public Integer numPartitions() {
            return numPartitions;
        }

        /**
         * Return the number of replicas to create or null if replicaAssignments is not null.
         */
        public Short replicationFactor() {
            return replicationFactor;
        }

        /**
         * Return a map from partition id to replica (broker) ids or null if numPartitions and
replicationFactor are
         * set instead.
         */
        public Map<Integer, List<Integer>> replicasAssignments() {
            return replicasAssignments;
        }

        /**
         * Return topic configs in the request, not including broker defaults. Broker configs are passed via
         * the {@code configure()} method of the policy implementation.
         */
        public Map<String, String> configs() {
            return configs;
        }

    }

    /**
     * Validate the request parameters and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the create request parameters for the provided topic do not satisfy this policy.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message. Note that
validation
     * failure only affects the relevant topic, other topics in the request will still be processed.
     *
     * @param requestMetadata the create request parameters for the provided topic.
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validate(RequestMetadata requestMetadata) throws PolicyViolationException;
}
```

Users will have to ensure that the policy implementation code is in the broker's classpath. Implementations should throw the newly introduced `PolicyViolationException` with an appropriate error message if the request does not fulfill the policy requirements. We chose a generic name for the only parameter of the `validate` method in case we decide to add parameters that are not strictly related to the topic (e.g. session information) in the future. The constructor of `RequestMetadata` is public to make testing convenient for users. Under normal circumstances, it should only be instantiated by Kafka. We chose to create separate API classes instead of reusing request classes to make it easier to evolve the latter.

The fact that topic creation can now fail due to custom policies raises new requirements at the protocol level:

1. We need to be able to send error messages back to the client, so we introduce an `error_message` string field in the `topic_errors` array of the `CreateTopics` response.
2. We need a new error code for `PolicyViolationException`, so we assign error code 44 to `POLICY_VIOLATION`.
3. For tools that allow users to create topics, a validation/dry-run mode where validation errors are reported but no creation is attempted is useful. A precedent for this exists in the Connect REST API. We introduce a `validate_only` boolean field in the `CreateTopics` request to enable this mode.

Both request and response versions are bumped to 1.

```
CreateTopics Request (Version: 1) => [create_topic_requests] timeout validate_only (new)
  create_topic_requests => topic num_partitions replication_factor [replica_assignment] [configs]
    topic => STRING
    num_partitions => INT32
    replication_factor => INT16
    replica_assignment => partition_id [replicas]
      partition_id => INT32
      replicas => INT32
    configs => config_key config_value
      config_key => STRING
      config_value => STRING
  timeout => INT32
  validate_only => BOOLEAN (new)

CreateTopics Response (Version: 1) => [topic_errors]
  topic_errors => topic error_code error_message (new)
    topic => STRING
    error_code => INT16
    error_message => NULLABLE_STRING (new)
```

Proposed Changes

During broker start-up, AdminManager will create a `CreateTopicPolicy` instance if `create.topic.policy.class.name` is defined. It will then pass the broker configs to the `configure` method.

When a create topics request is received, each topic will be processed in sequence. For each topic:

1. It will first perform the existing hardcoded request parameters validation (`numPartitions` and `replicationFactor` cannot be used at the same time as `replicaAssignments)` followed by `CreateTopicPolicy.validate` (if defined).
2. If validation fails, the `POLICY_VIOLATION` error code and error message will be added to the response (for this topic). Note that the error message will only be included if the request version is greater than or equal to 1.
3. If validation succeeds and `validateOnly` is true, we return the NONE error code for the topic without attempting topic creation. If `validateOnl` is false, topic creation will be attempted as usual.

Note that validation failure only affects the relevant topic, other topics in the request will still be processed. Also, it's worth mentioning that `validateOnly` doesn't guarantee that topic creation will succeed. Errors could still occur during the actual creation process.

During broker shutdown, `CreateTopicsPolicy.close` will be invoked.

As described in the previous section, we are proposing one policy config/interface per supported request type. The main advantage is that we can add additional configs in a compatible manner, but it also allows for modular policy implementations. Implementors also have the option of using a single implementation class if they wish, but multiple configs would still have to be set.

Compatibility, Deprecation, and Migration Plan

There is no compatibility impact as there is no change in behaviour unless the new config is used.

Future Work

1. As we add new policy interfaces, it could make sense to introduce an `admin.policy.class.name` config so that one could provide a single implementation for multiple policies without having to set multiple configs. Not clear if the cost is worth the benefit, however.
2. Classloader isolation: it may make sense to load user supplied implementations in a separate classloader to avoid dependency version clashes. This is currently being explored in the context of Connect and it may make sense to extend it to all user supplied code in the broker (including Authorizers).

Rejected Alternatives

1. A single config for an implementation of an interface with multiple `validate` methods: there would be no way to add methods without breaking binary compatibility until we move to Java 8.
2. A single config for an implementation of an abstract class with multiple `validate` methods, each with an empty implementation by default: this makes it easier to evolve when compared to rejected alternative 1, but provides less flexibility/modularity to implementors.
3. Restrict it via configs instead of pluggable interfaces: it would be clunky to provide configs for every requirement that users may have.
4. Extending Authorizer to perform validation: even though this could work, it's not particularly intuitive.