KIP-940: Broker extension point for validating record contents at produce time

Status

Current state: *"Under Discussion"*

Discussion thread: *https://lists.apache.org/thread/wyt...*

JIRA: *TBD here [TBD]*

Co-authored-by: **Adrian Preston** <*prestona@uk.ibm.com*> and **Edoardo Comar** <*ecomar@uk.ibm.com*>

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

Motivation

Using schemas to define the format of Kafka messages is generally considered a best practice, and has been widely adopted. Most aspects relating to the use of schemas can be managed outside of the core Kafka project and integrated using client key/value serializer/deserializer implementations. Typically these serializer/deserializers are coupled with, and maintained alongside a particular schema registry implementation, of which there are several that a user can choose from.

Unfortunately by only integrating via the serializer/deserializer client extension points, there is no way for an administrator to enforce the use of schemas. A single misconfigured client (e.g., a client using incorrect serializer/deserializers) can cause chaos by producing messages using an unexpected format, or without the constraints enforced by a schema. Such a client may prove difficult to track down and fix.

This KIP proposes a new broker-side extension point (a “record validation policy”) that can be used to reject records published by a misconfigured client. If a schema registry implementation chooses to provide its own record validation policy (maintained alongside its existing client serializer/deserializers) it can be used to prevent misconfigured clients from producing messages to Kafka.

Relationship to other KIPS

There are already a number of KIPS describing similar function. In our opinion, this suggests there is general interest in providing these capabilities. We have incorporated elements (and feedback) from these other KIPS:

KIP-686: API to ensure Record policy on the broker (Discussion stopped in 2020) - has similar intent, but proposed using PolicyViolationException for rejecting records, while we suggest reusing the InvalidRecordException from KIP-467 which maps to an error handled by the Producer. In addition, KIP-686 does not describe how it would handle compression.

KIP-729: Custom validation of records on the broker prior to append (2021)- has similar intent though it states that it will not work with compression - which is addressed in our KIP.

KIP-905: Broker interceptors (currently active) - is much broader in scope as it covers use-cases that include both mutating messages, and also intercepting client fetch requests. Our KIP can be viewed as a more narrowly focused subset of the function described in KIP-905, targeted at improving the experience of using schemas with Kafka.

KIP-467: Augment ProducerResponse error messaging for specific culprit records provides the foundation of the broker validation logic, and client external behavior, that this KIP build upon. KIP-467 is already implemented, with the exception of the client retries functionality.

Public Interfaces

New interface org.apache.kafka.server.RecordValidationPolicy

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

```java
/**
* An interface for enforcing a policy on the records that are accepted by this
* broker.
*
* A common use case is for validating that records contain the correct schema
* information.
*
* If the broker config <code>record.validation.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).
*
* If the broker config <code>record.validation.policy.class.path</code> is defined,
* the RecordValidationPolicy implementation and its dependent libraries will be loaded
* by a dedicated classloader which searches this class path before the Kafka broker class path.
* The syntax of this string parameter that of a Java class path.
*/

public interface RecordValidationPolicy extends Configurable, Closeable {

 /**
 * TopicMetadata describes the topic-partition a record is being produced to.
 */
 public interface TopicMetadata {
   TopicIdPartition topicIdPartition();

  /**
  * @return the value of the topic config <code>"record.validation.policy"</code>
  */
  String validationPolicy();
 }

/**
* HeaderProxy allows read-only access to the data in a record header
*/
public interface HeaderProxy {
  String key();


  /**
  * @return a read-only view on the header value
  */
  ByteBuffer value();
}

/**
* RecordProxy allows read-only access to the parts of a record that can be inspected by a validation policy.
*
* For efficiency, only the data required by the policy {@linkplain}RecordIntrospectionHints
* are guaranteed to be available to the policy.
*/
public interface RecordProxy {
  /**
  * @return a read-only list of header data
  */
  List<HeaderProxy> headers();


  /**
  * @return a read-only view on the record key
  */
  ByteBuffer key();
```

```
  /**
   * @return a read-only view on the record value
   */
  ByteBuffer value();
}

/**
* @throws InvalidRecordException when this policy rejects a record
*/
void validate(TopicMetadata topic, RecordProxy record) throws InvalidRecordException;

/**
* The parts of the record that a policy needs to inspect. This is used for
* iteration when using compression, so as to only decompress the minimum data
* necessary to perform the validation
*/
public interface RecordIntrospectionHints {
/**
   * @return whether the policy will need to access a record's headers
   */
boolean accessHeaders();

/**
* @return minimum number of bytes from the beginning of the record's key byte[]
* @return 0 key is not needed for policy
* @return -1 or Long.MAX_LONG for all
*/
long accessKeyBytes();

/**
* @return minimum number of bytes from the beginning of the record's value byte[]
* @return 0 key is not needed for policy
* @return -1 or Long.MAX_LONG for all
*/
long accessValueBytes();
}

/**
* @return hints describing the parts of the record that this policy
* needs to inspect.
*/
RecordIntrospectionHints getHints();
}
```

New Broker config

`record.validation.policy.class.name`

The fully qualified name of a class that implements a record validation policy. If no class name is defined then the default behavior is to perform no validation on the records sent to the broker.

Type: string

Default: null

Valid values:

Importance: low

Update mode: read-only

`record.validation.policy.class.path`

An optional Java class path for the `RecordValidationPolicy` implementation. If specified, the `RecordValidationPolicy` implementation and its dependent libraries will be loaded by a dedicated classloader which searches this class path before the Kafka broker class path. The syntax of this string parameter that of a Java class path.

Type: string

Default: null

Valid values:

Importance: low

Update mode: read-only

New Topic config

`record.validation.policy`

The record policy to use for the topic. This value should be meaningful to the record validation policy that the broker is configured with. If set to null (the default) then no validation will be performed for records sent to the topic irrespective of whether the broker is configured with a record validation policy class name.

Type: string

Default: null

Valid values:

Importance: low

Update mode: read-only

Proposed Changes

The current component `org.apache.kafka.storage.internals.log.LogValidator` already performs some validation of records before they are appended to a leader's log. For example, by ensuring that records produced to a compacted topic have a non-null key.

This KIP proposes extending the validation to include an optional call to a RecordValidationPolicy implementation. This call is only made if the broker is configured with a record validation policy class, and the topic the record is being produced to has a non-null record.validation.policy config. Consistently with KIP-467, if the policy rejects a record then the entire batch of records will be rejected.

Currently if a record is produced using a compression type that matches the compression setting of the topic it is being produced to, then the log validator logic is optimised to skip decompressing parts of the message that do not contain fields that it needs to inspect. This KIP introduces a `RecordIntrospectionHints` class to provide a description of the parts of the record a policy implementation needs to inspect. This metadata can be used by the broker to perform an analogous optimization, avoiding the need to de-compress parts of the record batch that will not be inspected by the validation policy.

Compatibility, Deprecation, and Migration Plan

- This KIP does not impact existing users, as the default record validation behavior is unchanged
- While this KIP is expected to apply only when message version >=2, there seem to be no increased implementation complexity to support older message versions.

Test Plan

This KIP implementation requires JUnit-style integration tests that can be added to the existing suites.

Rejected Alternatives

- We discarded the idea that the validation policy plugin class could be defined per-topic, as a single per-broker policy could act as a facade to different policies specific to topics (e.g. different registries)
- We discarded the idea to add an optional filed (eg schema id) to the kafka protocol Produce API, as that would require new clients not just serdes. Moreover, KIP-467 makes this approach look unnecessary.
- It remains possible to implement a Kafka Streams functionality that filters a topic that clients write to, into another topic with validated messages (and maybe a "dead letter" topic for invalid ones). Such an alternative approach doesn't provide any direct feedback to the client producer in the response. It also doesn't require the broker to execute third party code semantically coupled with the clients, at the price of having an extra "moving part" (the Streams app) which contains such logic. Moreover, the topic-to-schema mapping must map both input topic and destination topic to the same schema.
- We discarded having a boolean topic config to control per-topic enablement of the validation, when a policy is specified at the broker level, in favor of a string topic config whose value that can be consumed by the policy, as per the sample below.

Sample test policy

Here is a sample of a validation policy that we may use for the unit tests suite.

```
// Example RecordValidationPolicy implementation that checks record headers. Only records
// containing a header with a key that matches the 'record.validation.policy' property of
// the topic are accepted.
public class RequireHeaderValidationPolicy implements RecordValidationPolicy {
```

```
@Override
public void configure(Map<String, ?> configs) {
  // This example doesn't use any of the broker's configuration properties.
}

@Override
public void close() throws IOException {
  // This example doesn't need to perform any cleanup when the broker is shutdown.
}

@Override
public void validate(TopicMetadata topic, RecordProxy record) throws InvalidRecordException {
  // Do any of the record headers have a key matching the 'record.validation.policy'
  // property of the topic being produced to?
  if (!record.headers().stream().filter(h -> h.key().equals(topic.validationPolicy())).findFirst().isPresent()) {
  throw new InvalidRecordException(
   String.format("Topic %s requires records to contain a header with key: %s",
   topic.topicIdPartition().topic(), topic.validationPolicy()));
  }
}

@Override
public RecordIntrospectionHints getHints() {
// RequireHeaderValidationPolicy only requires access to the record headers.
// 0 is returned for the key/value portions of the record as this policy
// doesn't inspect these fields. This avoids any potential overhead in the
// broker parsing the record to find fields that the policy doesn't make use of.
return new RecordIntrospectionHints() {
  @Override
  public boolean accessHeaders() {
   return true;
  }

  @Override
  public long accessKeyBytes() {
   return 0;
  }

  @Override
  public long accessValueBytes() {
   return 0;
  }
};
}

}
```