KIP-145 - Expose Record Headers in Kafka Connect

- Status
- Motivation
- Public Interfaces
  - Connect Header and Headers Interfaces
  - Connect Records
  - Serialization and Deserialization
  - Configuration Properties
  - Converting Header Values
  - Transformations
    - HeaderFrom
    - HeaderTo
    - InsertHeader
    - DropHeaders
- Proposed Changes
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: *Accepted*

Discussion thread: here

JIRA: KAFKA-5142 - KIP-145 - Expose Record Headers in Kafka Connect

Released: 1.1.0

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

Motivation

KIP-82 introduced Headers into the core Kafka Product, and it would be advantageous to expose them in the Kafka Connect Framework. Whereas Kafka's headers are simple names with binary values, the Connect API already has a very useful layer for handling different types of data. Connect's header support should use string names like Kafka but values that are represented using the same types as used for Connect record keys and values. This will provide consistency with the rest of the Connect framework, and makes it easy for connectors and transforms to access, modify, and create headers on records.

Public Interfaces

The proposal discussed in this KIP is implemented in this pull request.

Kafka defines a header as having a string name and binary value, but the Connect will represent header values using the same mechanism used for record keys and values. Each header value may have a corresponding `Schema`, allowing connectors and transforms to work with header values, record keys, and record values in a consistent manner. Connect will define a `HeaderConverter` mechanism to serialize and deserialize header values in a manner similar to `Converter` framework, and such that the existing `Converter` implementations can also implement `HeaderConverter`. Since connectors and transformations from different vendors may be combined into a single pipeline, it is important that the different connectors and transformations can easily convert the header values from the original form into types that the connector and/or transform expect.

This KIP adds several new interfaces and changes a number of the existing public interfaces, and these are outlined below.

Note: *The code shown in this KIP excludes JavaDoc for brevity and clarity, but the proposed changes do include JavaDoc for all public APIs and methods.*

Connect Header and Headers Interfaces

A new `org.apache.kafka.connect.Header` interface will be added and used as the public API for a single header on a record. The interface defines simple getters for the key, the value, and the value's schema. These are immutable objects, and there are also methods to creating a new `Header` object with a different name or value.

```
package org.apache.kafka.connect.header;
public interface Header {

    // Access the key and value
    String key(); // never null
    Schema schema(); // may be null
    Object value(); // may be null

    // Methods to create a copy
    Header with(Schema schema, Object value);
    Header rename(String key);
}
```

A new `org.apache.kafka.connect.Headers` interface will also be added and used as the public API for the ordered list of headers for a record. This is patterned after Kafka client's `org.apache.kafka.common.header.Headers` interface as an ordered list of headers where multiple headers with the same name are allowed. The Connect `Headers` interface defines methods to access the individual `Header` objects sequentially and/or by name, and to get information about the number of `Header` objects. It also defines methods to add, remove, and retain `Header` objects using a variety of signatures that will be easy for connectors and transforms to use. And since multiple `Header` objects can have the same name, transforms need an easy way to modify and/or remove existing `Header` objects, and the `apply(HeaderTransform)` and `apply(String, HeaderTransform)` methods make it easy to use custom lambda functions to do this.

```
package org.apache.kafka.connect.header;
public interface Headers extends Iterable<Header> {

    // Information about the Header instances
    int size();
    boolean isEmpty();
    Iterator<Header> allWithName(String key);
    Header lastWithName(String key);

    // Add Header instances to this object
    Headers add(Header header);
    Headers add(String key, SchemaAndValue schemaAndValue);
    Headers add(String key, Object value, Schema schema);
    Headers addString(String key, String value);
    Headers addBoolean(String key, boolean value);
    Headers addByte(String key, byte value);
    Headers addShort(String key, short value);
    Headers addInt(String key, int value);
    Headers addLong(String key, long value);
    Headers addFloat(String key, float value);
    Headers addDouble(String key, double value);
    Headers addBytes(String key, byte[] value);
    Headers addList(String key, List<?> value, Schema schema);
    Headers addMap(String key, Map<?, ?> value, Schema schema);
    Headers addStruct(String key, Struct value);
    Headers addDecimal(String key, BigDecimal value);
    Headers addDate(String key, java.util.Date value);
    Headers addTime(String key, java.util.Date value);
    Headers addTimestamp(String key, java.util.Date value);

    // Remove and/or retain the latest Header
    Headers clear();
    Headers remove(String key);
    Headers retainLatest(String key);
    Headers retainLatest();

    // Create a copy of this Headers object
    Headers duplicate();

    // Apply transformations to named or all Header objects
    Headers apply(HeaderTransform transform);
    Headers apply(String key, HeaderTransform transform);

    interface HeaderTransform {
        Header apply(Header header);
    }
}
```

Connect Records

Every Kafka message contains zero or more header name-value pairs, and so the Connect record classes will be modified to have a non-null `Headers` object that can be modified in place. The existing `ConnectRecord` abstract class is the base class for both `SourceRecord` and `SinkRecord`, and will be changed to have the new `headers` field populated with a `ConnectHeaders` object. The signatures of all existing constructors and methods will be unmodified to maintain backword compatibility, but the existing constructor will populate the new `headers` field with a `ConnectHeaders` object. And, the `toString()`, `hashCode()`, and `equalTo(Object)` methods will be changed to make use of the new `headers` field.

A new constructor and several new methods will be added to this existing class:

```
package org.apache.kafka.connect.connector;
public abstract class ConnectRecord<R extends ConnectRecord<R>> {


   /* The following will be added to this class */

   private final Headers headers;
   public ConnectRecord(String topic, Integer kafkaPartition,
                        Schema keySchema, Object key,
                        Schema valueSchema, Object value,
                        Long timestamp, Iterable<Header> headers) {
       this(topic, kafkaPartition, keySchema, key, valueSchema, value, timestamp);
       if (headers == null) {
           this.headers = new ConnectHeaders();
       } else if (headers instanceof ConnectHeaders) {
           this.headers = (ConnectHeaders)headers;
       } else {
           this.headers = new ConnectHeaders(headers);
       }
   }

   public Headers headers() {
       return headers;
   }

   public abstract R newRecord(String topic, Integer kafkaPartition, Schema keySchema,
                               Object key, Schema valueSchema, Object value, Long timestamp,
                               Iterable<Header> headers);
}
```

The existing `SourceRecord` class will be modified to add a new constructor and to implement the additional `newRecord(...)` method. Again, the signatures of all existing constructors and methods will be untouched to maintain backward compatibility.

```
package org.apache.kafka.connect.source;
public class SourceRecord extends ConnectRecord<SourceRecord> {

    /* The following will be added to this class */

    public SourceRecord(Map<String, ?> sourcePartition, Map<String, ?> sourceOffset,
                        String topic, Integer partition,
                        Schema keySchema, Object key,
                        Schema valueSchema, Object value,
                        Long timestamp, Iterable<Header> headers) {
        super(topic, partition, keySchema, key, valueSchema, value, timestamp, headers);
        this.sourcePartition = sourcePartition;
        this.sourceOffset = sourceOffset;
    }


    @Override
    public SourceRecord newRecord(String topic, Integer kafkaPartition, Schema keySchema, Object key, Schema
valueSchema, Object value,
                                  Long timestamp, Iterable<Header> headers) {
        return new SourceRecord(sourcePartition, sourceOffset, topic, kafkaPartition, keySchema, key,
valueSchema, value, timestamp, headers);
    }
}
```

Likewise, the existing `SinkRecord` class will be modified to add a new constructor and to implement the additional `newRecord(...)` method. Again, the signatures of all existing constructors and methods will be untouched to maintain backward compatibility.

```
package org.apache.kafka.connect.sink;
public class SinkRecord extends ConnectRecord<SinkRecord> {

    /* The following will be added to this class */

    public SinkRecord(String topic, int partition,
                      Schema keySchema, Object key, Schema valueSchema, Object value, long kafkaOffset,
                      Long timestamp, TimestampType timestampType, Iterable<Header> headers) {
        super(topic, partition, keySchema, key, valueSchema, value, timestamp, headers);
        this.kafkaOffset = kafkaOffset;
        this.timestampType = timestampType;
    }

    @Override
    public SinkRecord newRecord(String topic, Integer kafkaPartition, Schema keySchema, Object key, Schema
valueSchema, Object value,
                                Long timestamp, Iterable<Header> headers) {
        return new SinkRecord(topic, kafkaPartition, keySchema, key, valueSchema, value, kafkaOffset(),
timestamp, timestampType, headers);
    }
}
```

Serialization and Deserialization

This proposal adds a new `org.apache.kafka.connect.storage.HeaderConverter` interface that is patterned after the existing `org.apache.kafka.connect.storage.Converter` interface except with header-specific method names and signatures.

```
package org.apache.kafka.connect.storage;
public interface HeaderConverter extends Configurable, Closeable {

    /**
     * Convert the header name and byte array value into a {@link Header} object.
     * @param topic the name of the topic for the record containing the header
     * @param headerKey the header's key; may not be null
     * @param value the header's raw value; may be null
     * @return the {@link SchemaAndValue}; may not be null
     */
    SchemaAndValue toConnectHeader(String topic, String headerKey, byte[] value);

    /**
     * Convert the {@link Header}'s {@link Header#valueAsBytes() value} into its byte array representation.
     * @param topic the name of the topic for the record containing the header
     * @param headerKey the header's key; may not be null
     * @param schema the schema for the header's value; may be null
     * @param value the header's value to convert; may be null
     * @return the byte array form of the Header's value; may be null if the value is null
     */
    byte[] fromConnectHeader(String topic, String headerKey, Schema schema, Object value);

    /**
     * Configuration specification for this set of header converters.
     * @return the configuration specification; may not be null
     */
    ConfigDef config();
}
```

Note that unlike `Converter`, the new `HeaderConverter` interface extends the `Configurable` interface that is now common for Connect interfaces that may have additional configuration properties.

It is possible for any existing implementation of `Converter` to also implement `HeaderConverter`, and all three existing `Converter` implementations in Connect will be changed accordingly to also implement this new interface by serializing/deserializing header values similarly to how they serialize /deserialize keys and values:

- `StringConverter`
- `ByteArrayConverter`

- `JsonConverter`

A new `HeaderConverter` implementation will be added to convert all built-in primitives, arrays, maps, and structs to and from string representations. Unlike the `StringConverter` that uses the `toString()` methods, the `SimpleHeaderConverter` uses JSON-like representation for primitives, arrays, maps, and structs, except for simple string values that are unquoted. This form corresponds directly to what many developers would think to serialize values as strings, and it allos the `SimpleHeaderConverter` to parse these any and all such values and most of the time to infer the proper schema. As such, this will be used for the default `HeaderConverter` used in the Connect worker.

The following table describes how these values will be persisted by the `SimpleHeaderConverter`.

Schema. Type	Description of string represesntation	Example
BOOLEAN	Either the `true` or `false` literals strings	
BYTE_ARRAY	Base64 encoded string representation of the byte array	
INT8	The string representation of a Java byte.	
INT16	The string representation of a Java short.	
INT32	The string representation of a Java int.	
INT64	The string representation of a Java long.	
FLOAT32	The string representation of a Java float.	
FLOAT64	The string representation of a Java double.	
STRING	The UTF-8 representation of the string, without surrounding quotes.	
ARRAY	A JSON-like representation of the array. Array values can be of any type, including primitives and non-primitives. Embedded strings are quoted. However, when parsing, the schema will only be inferred when all values are of the same type.	
MAP	A JSON-like representation of the map. Although most properly-created maps will have the same type of key and value, maps with any keys and values are also supported. Map values can be of any type, including primitives and non-primitives. Embedded strings are quoted. However, when parsing, the schema will be inferred only when the keys are strings and all values have the same type.	`{ "foo": "value", "bar": "strValue", "baz" : "other" }`
STRUCT	A JSON-like representation of the struct. Struct object can be serialized, but when deserialized will always be parsed as maps since the schema is not included in the serialized form.	`{ "foo": true, "bar": "strValue", "baz" : 1234 }`
DECIMAL	The string representation of the corresponding `java.math.BigDecimal`.	
TIME	The IOS-8601 representation of the time, in the format "HH:mm:ss.SSS'Z'".	16:31:05.387UTC
DATE	The ISO-8601 representation of the date, in the format "YYYY-MM-DD".	2017-05-21
TIMESTAMP	The ISO-8601 representation of the timestamp, in the format "YYYY-MM-DD'T'HH:mm:ss.SSS'Z'".	2017-05-21T16:31:05.387UTC

Configuration Properties

The Connect workers need to be configured to use a `HeaderConverter` implementation, and so one additional worker configuration named `header.converter` will be defined, defaulting to the `SimpleHeaderConverter`. A similar configuration property with the same name and default will be added to the connector configuration, allowing connectors to override the worker's header converter. Note that each Connector task will have its own header converter instance, just like the key and value converters.

Converting Header Values

Each `Header` has a value that can be used by sink connectors and simple message transforms. However, the type of the header's values depends on how the headers were created in the first place and how they were serialized and deserialized. A new set of conversion utility methods will be added to make it easy for SMTs and sink connectors to convert the header values into a type that it can easily use. These conversions may require both the original schema and value. The conversions to and from strings use the same mechanism described by the `SimpleHeaderConverter` above.

For example, an SMT or sink connector might expect a header value to be a long, and can use these utility methods to convert any numeric value (e.g., int, short, String, BigDecimal, etc.). Or, a sink connector might expect a `Timestamp` logical data type, so it can use the `Values.convertToTimestamp (s,v)` method to convert from any ISO-8601 formatted string representation of a timestamp or date, or number of millis past epoch represented as a long or string.

These utility methods can be used for header values or for any value in keys, values, or within Structs, arrays, and maps.

```
package org.apache.kafka.connect.data;
public class Values {

    // All methods return null when value is null, and throw a DataException
    // if the value cannot be converted to the desired type.
    // If the value is already the desired type, these methods simply return it.
    public static Boolean convertToBoolean(Schema schema, Object value) throws DataException {...}
    public static Byte convertToByte(Schema schema, Object value) throws DataException {...}
    public static Short convertToShort(Schema schema, Object value) throws DataException {...}
    public static Integer convertToInteger(Schema schema, Object value) throws DataException {...}
    public static Long convertToLong(Schema schema, Object value) throws DataException {...}
    public static Float convertToFloat(Schema schema, Object value) throws DataException {...}
    public static Double convertToDouble(Schema schema, Object value) throws DataException {...}
    public static String convertToString(Schema schema, Object value) {...}
    public static java.util.Date convertToTime(Schema schema, Object value) throws DataException {...}
    public static java.util.Date convertToDate(Schema schema, Object value) throws DataException {...}
    public static java.util.Date convertToTimestamp(Schema schema, Object value) throws DataException {...}
    public static BigDecimal convertToDecimal(Schema schema, Object value, int scale) throws DataException {...}

    // These only support converting from a compatible string form, which is the same
    // format used in the SimpleHeaderConverter described above
    public static List<?> convertToList(Object value) {...}
    public static Map<?, ?> convertToMap(Object value) {...}

    // Only supports returning the value if it already is a Struct.
    public static Struct convertToStruct(Object value) {...}
}
```

Transformations

Several new Simple Message Transformations (SMTs) will also be defined to make it easy for Connect users to move/copy headers into fields on the keys /values, and to create headers from existing fields on the keys/values.

HeaderFrom

This new transformation moves or copies fields in the key/value on a record into that record's headers. The transformation takes as inputs:

- `fields` - a comma-separated list of field names whose values are to be copied/moved to headers.
- `headers` - a comma-separated list of header names, in the same order as the field names listed in the `fields` configuration property.
- operation - either `move` if the fields are to be **moved**, or `copy` if the fields are to just **copied** and left on the record.

The `fields` and `headers` properties must contain the same number of literals.

The following is a sample configuration that moves two fields in the records' value into headers on the record, and removes the fields.

```
transforms=moveValueFieldsToHeader
transforms.moveValueFieldsToHeader.type=org.apache.kafka.connect.transforms.HeaderFrom$Value
transforms.moveValueFieldsToHeader.fields=txnId, address
transforms.moveValueFieldsToHeader.headers=txn.id, address
transforms.moveValueFieldsToHeader.operation=move
```

HeaderTo

This new transformation moves or copies headers on a record into fields in on that record's key/value. The transformation takes as inputs:

- `headers` - a comma-separated list of header names to copy/move
- `fields` - a comma-separated list of field names to be updated with the corresponding header value, in the same order as the field names listed in the `fields` configuration property.
- `operation` - either `move` if the headers are to be **moved**, or `copy` if the headers are to just **copied** and left on the record.

The `fields` and `headers` properties must contain the same number of literals.

The following is a sample configuration that copies one header into a field in the record's value, but leaves the header as-is.

```
transforms=copyHeaderToValueField
transforms.copyHeaderToValueField.type=org.apache.kafka.connect.transforms.HeaderTo$Value
transforms.copyHeaderToValueField.fields=txnId
transforms.copyHeaderToValueField.headers=txn.id
transforms.copyHeaderToValueField.operation=copy
```

InsertHeader

This new transformation adds or inserts a header to each record. The transformation takes as inputs:

- `header` - the name of the header
- `value.literal` - the literal value that is to be set as the header value on all records.

The following is a sample configuration that sets the `app.id` header with a literal string value of `best-app-ever` on every record:

```
transforms=insertAppIdHeader
transforms.insertAppIdHeader.type=org.apache.kafka.connect.transforms.InsertHeader
transforms.insertAppIdHeader.header=app.id
transforms.insertAppIdHeader.value.literal=best-app-ever
```

DropHeaders

This new transformation drops one or more headers from the record. The transformation takes as inputs:

- `headers` - a comma-separated list of header names to remove from every record.

```
transforms=dropAppIdHeader
transforms.dropAppIdHeader.type=org.apache.kafka.connect.transforms.DropHeaders
transforms.dropAppIdHeader.header.names=app.id, txn.id
```

Proposed Changes

In addition to the public interface changes defined above, we will also add implementation classes for the new public interfaces.

The `org.apache.kafka.connect.header.ConnectHeader` implements the `Header` interface, and the constructor will be protected to prevent users from explicitly instantiating this class. Instead, connectors and transforms can instantiate the `org.apache.kafka.connect.header.ConnectHeaders` implementation class and use the methods to add/modify/remove/transform the individual `Header` objects. Note that the `ConnectHeaders` implementation class lazily creates internal data structures, so that instances are by default very lightweight.

Likewise, the `WorkerTask` class will be updated to create the correct type of `HeaderConverter` configured through the worker or connector configuration. The `WorkerSourceTask` will be changed to pass any headers supplied by the source connector into the Kafka headers for the message. The `WorkerSinkTask` will also change to process the headers when it processes each record, creating a `ConnectHeaders` object for the message's headers.

See this pull request for more details on the proposed changes.

Compatibility, Deprecation, and Migration Plan

All changes are binary-compatible, since this proposal changes no existing method signatures and only adds new classes as well as new methods and constructors to existing classes. Existing Connect worker configuration files will also continue to work, since the only new worker configurations all have default values. Existing connector, transformation, and converter implementations need change, and headers will be passed through unmodified. The connector and transform implementations will have to be modified to make use of headers.

Rejected Alternatives

The current proposal is fairly simple to understand, but the implementation is fairly large. This was deemed to be the best solution overall, since it offered the most simple API for *using* headers in connectors and transformation. Some other alternatives that were considered included:

1. Reuse the existing Kafka `Header` and `Headers` interfaces – This simple alternative would represent the header values only as opaque binary arrays, making it very difficult for Connect users to combine into data pipelines several connectors and transformations from multiple developers. It also would mean that the keys and values in Connect records are defined with schemas and various value types, but header values are handled very differently.
2. Reuse the existing Kafka `Header` and `Headers` interfaces with utility methods – Similar to the previous alternative, except a family of utility methods would help connector and transform implementations work with headers. This was not clean, and still required connector and transform implementations to know how to serialize and deserialize the header values.
3. Support only string header values – This option is easy to use, but significantly less flexible than Kafka's header support.
4. Simpler serialization and deserialization – Modest header interfaces, with limited/simpler serialization and deserialization logic.
5. Conversion methods directly on the `Header` interface, rather than the `Values.convertToX` utility methods. Having the conversion methods on `Header` would be easier to use for headers, but the static utility methods serve the same purpose yet can be used anywhere and for any values.

See the history of this page for details on several earlier approaches.