CCP Understanding Parsers 2.0.1

Understanding Parsing

Date of publish: 2017-11-06

https://docs.cloudera.com/

Legal Notice

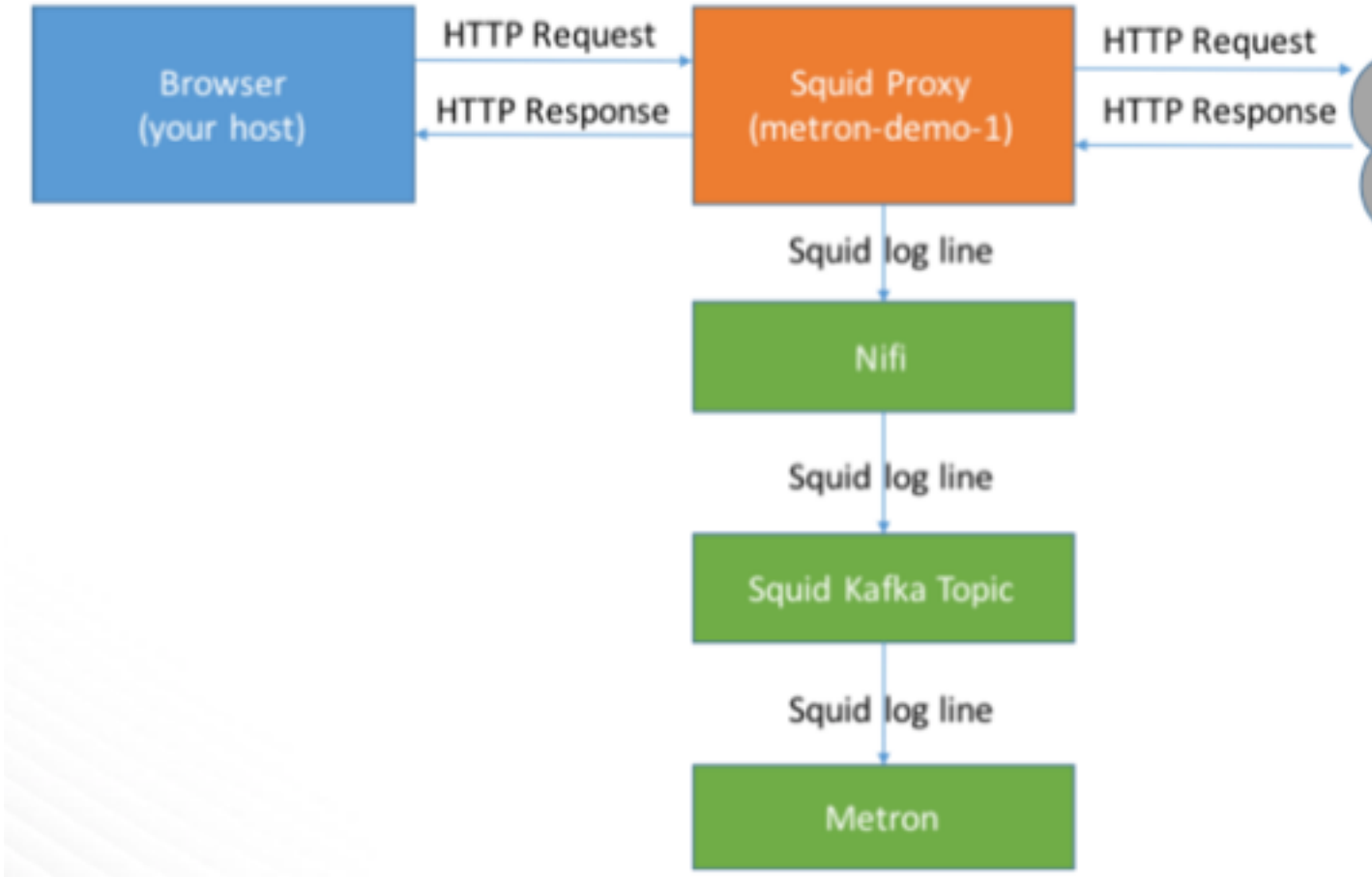
Contents

Parser Overview

Parsers are pluggable components that transform raw data (textual or raw bytes) into JSON messages suitable for downstream enrichment and indexing.

Data flows through the parser bolt via Apache Kafka and into the enrichments topology in Apache Storm.

For example, for a Squid parser, NiFi ingests the contents of the Squid proxy access log, the parser transforms the contents of the log, converts it to json, and inserts it into a Squid Kafka topic, which is then passed on to Metron.



CCP supports two types of parsers: general purpose and Java.

Errors are collected with the context of the error (for example, stacktrace) and the original message causing the error and are sent to an error queue. Invalid messages as determined by global validation functions are also treated as errors and sent to an error queue.

Java Parsers

The Java parser is written in Java and conforms with the MessageParser interface. This kind of parser is optimized for speed and performance and is built for use with higher-velocity topologies.

Java parsers are not easily modifiable; to make changes to them, you must recompile the entire topology.

Currently, the Java adapters included with CCP are as follows:

- org.apache.metron.parsers.ise.BasicIseParser
- org.apache.metron.parsers.bro.BasicBroParser
- org.apache.metron.parsers.sourcefire.BasicSourcefireParser
- org.apache.metron.parsers.lancope.BasicLancopeParser
- org.apache.metron.parsers.syslog.Syslog5424Parser
- org.apache.metron.parsers.syslog.Syslog3164Parser
- org.apache.metron.parsers.cef.CEFParser
- org.apache.metron.parsers.leef.LEEFParser

General Purpose Parsers

The general-purpose parser is primarily designed for lower-velocity topologies or for quickly setting up a temporary parser for a new telemetry.

General purpose parsers are defined using a config file, and you need not recompile the topology to change them. CCP supports two general purpose parsers: Grok and CSV.

Grok parser

The Grok parser class name (parserClassName) is org.apache.metron,parsers.GrokParser.

The Grok parser supports either one line to parse per incoming message, or incoming messages with multiple log lines, and will produce a json message per line

Grok has the following entries and predefined patterns for parserConfig:

grokPath	The path in HDFS (or in the Jar) to the grok statement. By default attempts to load from HDFS, then falls back to the classpath, and finally throws an exception if unable to load a pattern.
patternLabel	The pattern label to use from the Grok statement.
multiLine	The raw data passed in should be handled as a long with multiple lines, with each line to be parsed separately. This setting's valid values are true or false. The default if unset is false. When set, the parser will handle multiple lines with successfully processed lines emitted normally, and lines with errors sent to the error topic.
timestampField	The field to use for timestamp. If your data does not have a field exactly named "timestamp" this field is required, otherwise the record will not pass validation. If the timestampField is included in the list of timeFields, it will first be parsed using the provided dateFormat.
timeFields	A list of fields to be treated as time.
dateFormat	The date format to use to parse the time fields. Default is "yyyy-MM-dd HH:mm:ss.S z".
timezone	The timezone to use. UTC is the default.

CSV Parser

The CSV parser class name (parserClassName) is org.apache.metron.parsers.csv.CSVParser

CSV has the following entries and predefined patterns for parserConfig:

timestampFormat	The date format of the timestamp to use. If unspecified, the parser assumes the timestamp is starts at UNIX epoch.
columns	A map of column names you wish to extract from the CSV to their offsets. For example, { 'name' : 1,'profession' : 3} would be a column map for extracting the 2nd and 4th columns from a CSV.
separator	The column separator. The default value is ",".

JSON Map Parser

The JSON parser class name (parserClassName) is org.apache.metron.parsers.csv.JSONMapParser

JSON has the following entries and predefined patterns for parserConfig:

mapStrategy	A strategy to indicate how to handle multi-dimensional Maps. This is one of:

DROP	Drop fields which contain maps
UNFOLD	Unfold inner maps. So { "foo" : { "bar" : 1} } would turn into {"foo.bar" : 1}
ALLOW	Allow multidimensional maps
ERROR	Throw an error when a multidimensional map is encountered

timestamp	This field is expected to exist and, if it does not, then current time is inserted.
jsonQuery	If this JSON query string is present, the result of the query will be a list of messages. This is useful if you have a JSON document that contains a list or array of messages embedded in it, and you do not have another means of splitting the message.
wrapInEntityArray	This setting's valid values are true or false. If jsonQuery is present and this flag is present and set to "true", the incoming message will be wrapped in a JSON entity and array. for example: {"name":"value"},{"name2","value2"} will be wrapped as {"message" : [{"name":"value"}, {"name2","value2"}]}. This is using the default value for wrapEntityName if that property is not set.

wrapEntityName	Sets the name to use when wrapping JSON using wrapInEntityArray. The jsonpQuery should reference this name. Only applicable if jsonpQuery and wrapInEntityArray are specified.
timestamp	A field called timestamp is expected to exist and, if it does not, then current time is inserted.
overrideOriginalString	A boolean setting that will change the way original_string is handled by the parser. The default value of false uses the global functionality that will append the unmodified original raw source message as an original_string field. This is the recommended setting. Setting this option to true will use the individual substrings returned by the json query as the original_string. For example, a wrapped map such as {"foo" : [{"name":"value"}, {"name2","value2"}]} that uses the jsonpQuery, $.foo, will result in 2 messages returned. Using the default global original_string strategy, the messages returned would be: • { "name" : "value", "original_string" : "{\"foo\" : [{\"name\":\"value\"},{\"name2\",\"value2\"}]}} • { "name2" : "value2", "original_string" : "{\"foo\" : [{\"name\":\"value\"},{\"name2\",\"value2\"}]}} Setting this value to true would result in messages with original_string set as follows: • { "name" : "value", "original_string" : "{\"name \":\"value\"}} • { "name" : "value", "original_string" : "{\"name2\":\"value2\"}} One final important point to note, and word of caution about setting this property to true, is about how JSON PQuery handles parsing and searching the source raw message - it will NOT retain a pure raw sub-message. This is due to the JSON libraries under the hood that normalize the JSON. The resulting generated original_string values may have a different property order and spacing. For example, { "foo" :"bar" , "baz":"bang"} would end up with an original_string that looks more like { "baz" : "bang", "foo" : "bar" }.

Regular Expressions Parser

recordTypeRegex	A regular expression to uniquely identify a record type.
messageHeaderRegex	A regular expression used to extract fields from a message part which is common across all the messages.
convertCamelCaseToUnderScore	If this property is set to true, this parser will automatically convert all the camel case property

names to underscore separated. For example, following conversions will automatically happen:

```
ipSrcAddr -> ip_src_addr
ipDstAddr -> ip_dst_addr
ipSrcPort -> ip_src_port
```

Note this property may be necessary, because java does not support underscores in the named group names. So in case your property naming conventions requires underscores in property names, use this property.

fields A json list of maps containing a record type to regular expression mapping.

A complete configuration example looks like:

```
"convertCamelCaseToUnderScore": true,
"recordTypeRegex": "kernel|syslog",
"messageHeaderRegex": "(<syslogPriority>(<=^&lt;)\\d{1,4}(?=>)).*?
(<timestamp>(<=>)[A-Za-z] {3}\\s{1,2}\\d{1,2}\\s\\d{1,2}:\\d{1,2}:\\d{1,2}(?
=\\s)).*?(<syslogHost>(<=\\s).*?(?=\\s))",
"fields": [
  {
    "recordType": "kernel",
    "regex": ".*(<eventInfo>(<=\\]|\\w\\:).*?(?=$))"
  },
  {
    "recordType": "syslog",
    "regex": ".*(<processid>(<=PID\\s=\\s).*?(?=\\sLine)).*(<filePath>(<=64\
\s)\/([A-Za-z0-9_-]+\/)+(?=\\w))        (<fileName>.*?(?=
\")).*(<eventInfo>(<=\").*?(?=$))"
  }
]
```

Note: messageHeaderRegex and regex (within fields) can be specified as lists also. For example:

```
"messageHeaderRegex": [
  "regular expression 1",
  "regular expression 2"
]
```

Where:

regular expression 1 Valid regular expressions that may have named groups and which would be extracted into fields. This list will be evaluated in order until a matching regular expression is found.

messageHeaderRegex Run on all the messages. All messages are expected to contain the fields which are being extracted using the messageHeaderRegex. messageHeaderRegex is a sort of HCF (highest common factor) in all messages.

recordTypeRegex can be a more advanced regular expression containing named groups. For example:

```
"recordTypeRegex": "(<process>(<=\s)\b(kernel|syslog)\b(?=\[|:))"
```

All the named groups will be extracted as fields.

Though having named group in recordType is completely optional, you might want to extract named groups in recordType for following reasons:

- Because recordType regular expression is already getting matched and you are paying the price for a regular expression match already, you can extract certain fields as a by product of this match.
- The recordType field is probably common across all the messages. So, having it extracted in the recordType (or messageHeaderRegex) would reduce the overall complexity of regular expressions in the regex field.

regex within a field can also be a list of regular expressions. In this case all regular expressions in the list will be matched. Once a full match is found, remaining regular expressions are ignored.

```
"regex":  [ "record type specific regular expression 1",
            "record type specific regular expression 2"]
```

timestamp

Because this parser is a general purpose parser, it will populate the timestamp field with current UTC timestamp. Actual timestamp value can be overridden later using stellar. For example in case of syslog timestamps, you can use following stellar construct to override the timestamp value. Let us say you parsed actual timestamp from the raw log:

```
<38>Jun 20 15:01:17 hostName
 sshd[11672]: Accepted publickey for
 prod from 55.55.55.55 port 66666
 ssh2

syslogTimestamp="Jun 20 15:01:17"
```

Then something like the following can be used to override the timestamp:

```
"timestamp_str": "FORMAT('%s%s%s',
 YEAR(),' ',syslogTimestamp)",
"timestamp":"TO_EPOCH_TIMESTAMP(timestamp_str,
 'yyyy MMM dd HH:mm:ss' )"
```

Or, if you want to factor in the timezone:

```
"timestamp":"TO_EPOCH_TIMESTAMP(timestamp_str,
 timestamp_format, timezone_name )"
```

Parser Message Routing

Parser messages are routed to the Kafka enrichment topic by default.

You can change the output topic with the output_topic option when starting the parser topology or with the outputTopic parser configuration setting. The order of precedence from highest to lowest is as follows:

- Parser start script option
- Parser configuration setting
- Default enrichments topic

You can also route the message to other locations besides Kafka with the writerClassName parser configuration setting. Messages can be routed independently for each sensor type when configured with parser configuration settings.

Parser Configuration

The configuration for the various parser topologies is defined by JSON documents stored in ZooKeeper.

The JSON document consists of the following attributes:

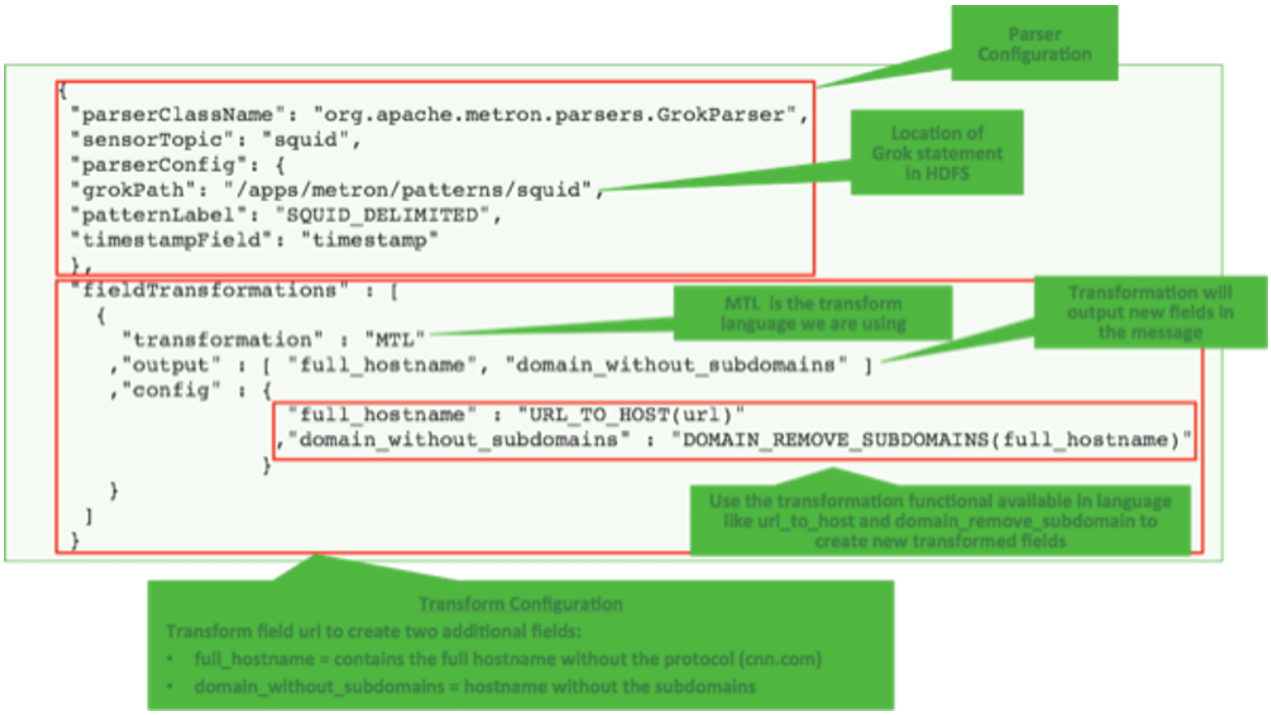
parserClassName	The fully qualified class name for the parser to be used.
sensorTopic	The Kafka topic to send the parsed messages to.
parserConfig	A JSON Map representing the parser implementation specific configuration.
fieldTransformations	An array of complex objects representing the transformations to be done on the message generated from the parser before writing out to the Kafka topic. The fieldTransformations is a complex object which defines a transformation that can be done to a message. This transformation can perform the following: • Modify existing fields to a message • Add new fields given the values of existing fields of a message • Remove existing fields of a message

Example: fieldTransformation Configuration

The fieldTransformation is a complex object which defines a transformation that can be done to a message.

In this example, the host name is extracted from the URL by way of the URL_TO_HOST function. Domain names are removed by using DOMAIN_REMOVE_SUBDOMAINS, thereby creating two new fields (full_hostname and domain_without_subdomains) and adding them to each message.

Configuration File with Transformation Information



The format of a fieldTransformation is as follows:

input

An array of fields or a single field representing the input. This is optional; if unspecified, then the whole message is passed as input.

output

The outputs to produce from the transformation. If unspecified, it is assumed to be the same as inputs.

transformation

The fully qualified class name of the transformation to be used. This is either a class which implements FieldTransformation or a member of the FieldTransformations enum.

config

A String to Object map of transformation specific configuration.

CCP currently implements the following fieldTransformations options:

REMOVE

This transformation removes the specified input fields. If you want a conditional removal, you can pass a Metron Query Language statement to define the conditions under which you want to remove the fields.

The following example removes field1 unconditionally:

```
{
...
    "fieldTransformations" : [
          {
            "input" : "field1"
          , "transformation" :
 "REMOVE"
```

```
          }
                        ]
}
```

The following example removes field1 whenever field2 exists and has a corresponding value equal to 'foo':

```
{
...
  "fieldTransformations" : [
          {
            "input" : "field1"
          , "transformation" :
 "REMOVE"
          , "config" : {
              "condition" :
 "exists(field2) and field2 ==
 'foo'"
                       }
          }
                        ]
}
```

IP_PROTOCOL

This transformation maps IANA protocol numbers to consistent string representations.

The following example maps the protocol field to a textual representation of the protocol:

```
{
...
    "fieldTransformations" : [
          {
            "input" : "protocol"
          , "transformation" :
 "IP_PROTOCOL"
          }
                        ]
}
```

STELLAR

Io

This transformation executes a set of transformations expressed as Stellar Language statements.

The following example adds three new fields to a message:

utc_timestamp	The UNIX epoch timestamp based on the timestamp field, a dc field which is the data center the message comes from and a dc2tz map mapping data centers to timezones.

url_host	The host associated with the url in the url field.
url_protocol	The protocol associated with the url in the url field.

```
{
...
    "fieldTransformations" : [
          {
           "transformation" :
 "STELLAR"
          ,"output" :
 [ "utc_timestamp", "url_host",
 "url_protocol" ]
          ,"config" : {
            "utc_timestamp" :
 "TO_EPOCH_TIMESTAMP(timestamp,
 'yyyy-MM-dd
HH:mm:ss', MAP_GET(dc, dc2tz,
 'UTC') )"
          ,"url_host" :
 "URL_TO_HOST(url)"
          ,"url_protocol" :
 "URL_TO_PROTOCOL(url)"
                      }
          }
                      ]
   ,"parserConfig" : {
      "dc2tz" : {
                "nyc" : "EST"
               ,"la" : "PST"
               ,"london" : "UTC"
                }
    }
}
```

Note that the dc2tz map is in the parser config, so it is accessible in the functions.