Monitoring Elasticsearch with Signals Alerting

Cliff Staley · 2020-07-13

Elasticsearch exposes plenty of useful information about the cluster health through its REST API. This data becomes even more helpful once we know how it changes over time. In this post, we create a Kibana dashboard monitoring the cluster health. The index, used within the dashboard, will be created with Search Guard Signals and filled with data from the Elasticsearch cluster health and nodes stats endpoints.

Input Data

We will use two HTTP endpoints for collecting monitoring data:

- Cluster Health - with general cluster information like status or number of nodes,
- Nodes Stats - with detailed statistics regarding each node.

Both endpoints expose plenty of data. Storing all the information each minute could be space consuming; thus, we decide to grab only the following fields that we find useful for this demo:

- Cluster Status - red, yellow or green,
- Number of data nodes within a cluster,
- Number of non-deleted documents within each node stats (indices.docs.count),
- Time spent on indexing operations (indices.indexing.index*time*in_millis),
- Total number of query operations (indices.search.query_total),
- Time spent on performing the queries (indices.search.query*time*in_millis).

Indexing monitoring data

We start with a fresh installation of Elasticsearch and Kibana, both with Search Guard plugins enabled. In Kibana, we create a new watch and switch the watch definition into *block mode*. We will add the checks that fetch data from the REST endpoints and transform it into a structure that fits Elasticsearch.

The first check will be of a type *HTTP* and will fetch cluster health data. In our demo, we use basic authentication with the user and password provided. Please refer to Signals documentation to find more about authenticating Signals HTTP Inputs and TLS configuration. For the sake of simplicity, in the code below we use the *trust_all* TLS flag, which should of course not be used in production:

```
{
  "type": "http",
  "request": {
    "method": "GET",
    "url": "https://localhost:9200/_cluster/health",
    "auth": {
      "type": "basic",
      "username": "admin",
      "password": "admin"
    }
  },
  "tls": {
    "trust_all": true
  }
}
```

Then we start transforming the collected data and create a *HashMap* variable named *doc* that will be later inserted as a document into the Elasticsearch cluster. We fill *doc* with cluster status and a number of existing data nodes, that were fetched from the cluster health endpoint. We also add a *@timestamp* field with a value equal to execution_time. The result of this transformation will be stored as *doc* variable in a watch context.

```
{
  "type": "transform",
  "name": "target_document",
  "source": """
    def doc = new HashMap();
    doc['status'] = data['status'];
    doc['number_of_data_nodes'] = data['number_of_data_nodes'];
    doc['@timestamp'] = execution_time;

    return doc;
""",
  "target": "doc"
}
```

Fetching node stats can be done similarly to cluster health. We are interested only in gathering indices stats, so we can limit the number of returned stats by adding extra parameter to a URL - *?metric=indices*.

```
{
  "type": "http",
  "request": {
    "method": "GET",
    "url": "https://localhost:9200/_nodes/stats?metric=indices",
    "auth": {
      "type": "basic",
      "username": "admin",
      "password": "admin"
    }
  },
  "tls": {
    "trust_all": true
  },
  "target": "stats"
}
```

Note that we specify the *target* field within the check. When doing so, the HTTP response will be available in further actions under *data.stats*. If we did not specify it, it would have been directly kept as a data variable, and data gathered in previous steps would have been overwritten.

Nodes stats return statistics per each node of the cluster. We need to create an extra transformation that aggregates data from multiple nodes:

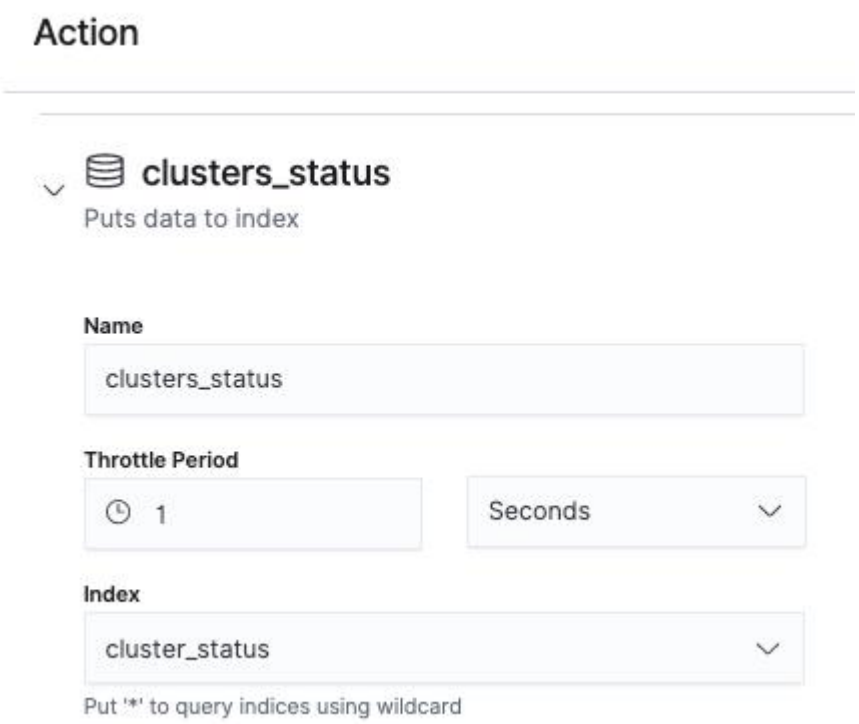
```
data.doc['docs_count'] = 0;
data.doc['index_time_in_millis'] = 0;
data.doc['query_total'] = 0;
data.doc['query_time_in_millis'] = 0;

for (node in data['stats']['nodes'].keySet()) {
  def stats = data['stats']['nodes'][node]['indices'];

  data.doc['docs_count'] += stats['docs']['count'];
  data.doc['index_time_in_millis'] += stats['indexing']['index_time_in_millis'];
  data.doc['query_total'] += stats['search']['query_total'];
  data.doc['query_time_in_millis'] += stats['search']['query_time_in_millis'];
}

return data.doc
```

The last step is to define an action that puts the data into Elasticsearch. This only requires providing an action name and index name where the data is stored.



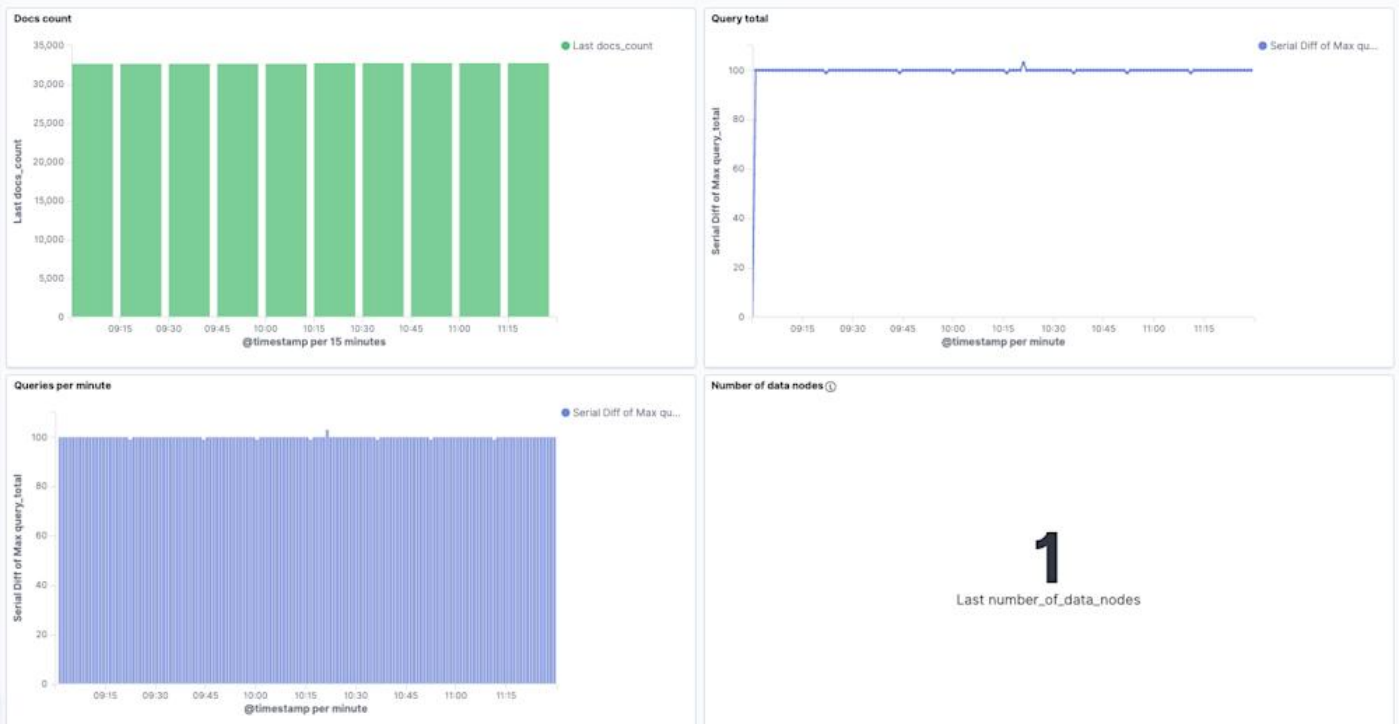
Visualizing monitoring data

Once a watch is activated and saved, it gathers monitoring data every minute or according to the interval defined within the watch. We can create a Kibana Index Pattern and discover data in Kibana:

#	docs_count	32,656
#	index_time_in_millis	1,015
#	number_of_data_nodes	1
#	query_time_in_millis	2,461
#	query_total	14,162
t	status	yellow

Let us now prepare some visualizations. Metrics like *query_total* and *querytimein_millis* are cumulative, which means they grow with time. A common technique to visualize such data is subtracting time series from itself at different time. In other words, if we subtract from a query*total that is now, a query*total an hour ago, we will know the number of queries that were run last hour. The technique is also known as Serial Differencing Aggregation.

This can lead us to a visualization like the one below:



Where to go next

- Read our getting started guide that comes with examples and demo watches
- Read more about the Kibana blocks mode for easy watch setup
- If you have questions, head over to the Signals section on our forum

Image: Shuttestock / Graphicfarm

This article was published on the Search Guard blog: https://search-guard.com/blog/elasticsearch-alerting-monitoring/
Search Guard is Security and Alerting for Elasticsearch and Kibana — search-guard.com · Start a free trial