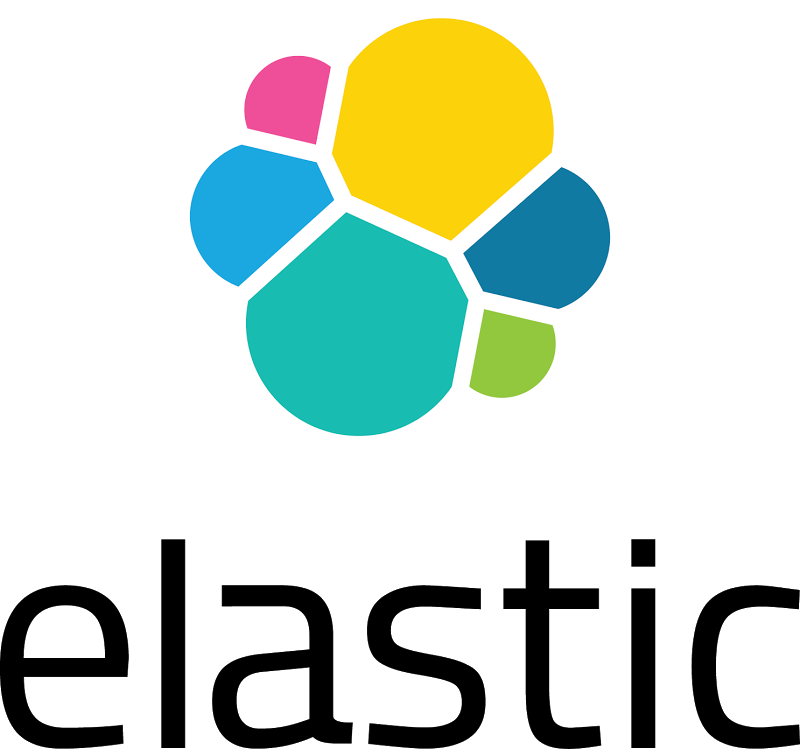
Alerting 5.5 Hands On Lab 

## Assumptions:

1. You have **Metricbeat** installed on your laptop. You can download it here: <https://www.elastic.co/downloads/beats/metricbeat>.

## Goals:

1. To understand how to configure and validate X-Pack Alerting.

2. To deploy some simple system health check alerts.

## Step 1: Update Elasticsearch configuration

|  |  |
| --- | --- |
| 1. | Find the IP address of your VM and open a browser to it. Make sure you log in as the elastic user. |
| 2. | Copy the contents of xpack-alerting.txt file and paste it into Dev Tools > Console.  This file contains all the commands that we will use throughout these exercises. |
| 3. | Find Step 1 and execute the command under 1.1.  GET \_xpack/watcher/stats  It should return the following, which indicates that X-Pack Alerting is successfully installed and available:  {  "watcher\_state": "started",  "watch\_count": 0,  "execution\_thread\_pool": {  "queue\_size": 0,  "max\_size": 0  },  "manually\_stopped": false  } |

## Step 2: Excessive shard count alert

With X-Pack Alerting installed and ready, we are going to create a relatively simple alert that will notify us when the total number of shards in the cluster exceeds a fixed threshold.

|  |  |
| --- | --- |
| 1. | In Console, execute the command under 2.1 and get familiar with the output from the cluster stats API.  GET \_cluster/stats  Locate indices.shards.total in the structure and note what it is set to. |
| 2. | The first version of the watch we will deploy can be found under step 2.2:  PUT \_xpack/watcher/watch/cluster\_shard\_count  {  "metadata": {  "max\_shards": 30  },  "trigger": {  "schedule": { "interval": "1m" }  },  "input": {  "http" : {  "request" : {  "host" : "localhost",  "port" : 9200,  "scheme": "https",  "auth": {  "basic": {  "username": "elastic",  "password": "changeme"  }  },  "path" : "/\_cluster/stats"  }  }  }  }  This watch uses an HTTPS request to the local node against the cluster stats API once every minute. The result is recorded, but no action or condition has so far been configured.  Execute this request. The response should be something like the following:  {  "\_id": "cluster\_shard\_count",  "\_version": 1,  "created": true  } |
| 3. | The results of watches can be found in the .watcher-history-\* index. Examine the results of the watch that was created through the command under 2.3:  GET .watcher-history-\*/\_search  {  "query": {  "match\_all": {}  },  "sort": { "trigger\_event.triggered\_time": "desc"}  }  Notice that the password that was submitted as the watch was created is not present in the results. |
| 4. | The next step is to add a condition and an action. In this example (see step 2.4) we will log a warning message only when the fixed shard threshold is exceeded. Add the following to the initial watch:  "condition" : {  "compare" : { "ctx.payload.indices.shards.total" : { "gt" : "{{ctx.metadata.max\_shards}}" } }  },  "actions" : {  "log\_warn" : {  "logging" : {  "text" : "The current number of shards in the cluster ({{ctx.payload.indices.shards.total}}) exceeds the set threshold ({{ctx.metadata.max\_shards}}).",  "level": "warn"  }  }  }  Execute the command under step 2.4, which should return something like this:  {  "\_id": "cluster\_shard\_count",  "\_version": 27,  "created": false  } |
| 5. | Inspect the result of the watch through the command under step 2.3. Verify that the executed watch did have the newly added condition and action. Verify that the watch should not have executed as the condition was not met by examining the state of the watch, which should be "execution\_not\_needed". |
| 6. | Now create a new index with 20 shards. The command is available under step 2.5:  PUT test  {  "settings" : {  "index" : {  "number\_of\_shards" : 20,  "number\_of\_replicas" : 0  }  }  }  This should return:  {  "acknowledged": true,  "shards\_acknowledged": true  } |
| 7. | Wait for around 30 seconds and then run the command under step 2.3 and verify that the watch executed and that the condition was met. Then delete the index:  DELETE test |
| 8. | Having a log entry created every time the watch executes might be too much, so let us modify the watch and throttle the logging action so that a log message is printed at most once every 5 minutes. This can be done per action as follows:  "actions" : {  "log\_warn" : {  "throttle\_period": "5m",  "logging" : {  "text" : "The current number of shards in the cluster ({{ctx.payload.indices.shards.total}}) exceeds the set threshold ({{ctx.metadata.max\_shards}}).",  "level": "warn"  }  }  }  Deploy this revised version through the command under step 2.7. Then create another index through the command under step 2.5.  Verify that the watch is triggering every minute.  GET .watcher-history-\*/\_search  {  "query": {  "match\_all": {}  },  "sort": { "trigger\_event.triggered\_time": "desc"}  }  Check the execution status though the command under step 2.3 and verify that the status shows up as "execution\_not\_needed". |
| 9. | Deactivate the watch though the following command (step 2.8):  PUT \_xpack/watcher/watch/cluster\_shard\_count/\_deactivate  Verify the watch is no longer being executed. |

## Step 3: Alert on process existence

For the next exercise we are going to use Metricbeat to generate some data. This will require you to have Metricbeat installed and running on your laptop. Metricbeat sends data to Elasticsearch about processes that are running on your laptop, with information like the process name and ID.

|  |  |
| --- | --- |
| 1. | If you haven’t already done so, please install Metricbeat now by downloading it from: <https://www.elastic.co/downloads/beats/metricbeat>.  Before we can start sending metrics to our VM, we have to edit metricbeat.yml to tell it where to send events. Open up metricbeat.yml and go to line 71 as shown below.  71 #---- Elasticsearch output --------------------   72 output.elasticsearch:   73   # Array of hosts to connect to.   74   hosts: ["localhost:9200"]   75   76   # Optional protocol and basic auth credentials.   77   #protocol: "http"   78   #username: "elastic"   79   #password: "changeme"   80  We need to change line 74 so that we send events to our VM. Go back to the web page where we initially spun up our VM and find the IP address.    We need to copy/paste the IP address of our VM into the metricbeat.yml file. For example, my VM’s IP address is 54.204.162.23. So, I’ll change line 74 by removing “localhost:9200” and inserting “54.204.162.23:9200”.  Next, uncomment lines 78-79. The end result should look something like this:  71 #---- Elasticsearch output --------------------   72 output.elasticsearch:   73   # Array of hosts to connect to.   74   hosts: ["54.204.162.23:9200"]   75   76   # Optional protocol and basic auth credentials.   77   #protocol: "http"   78 username: "elastic"   79   password: "changeme"   80  Now save and exit the file. Next we want to start Metricbeat. The start sequence is different depending on the platform on which we are running.  Mac:  1. sudo chown root metricbeat.yml  2. ./scripts/import\_dashboards –es http://<IP>:9200 -user elastic –pass changeme  3. sudo ./metricbeat –e –c metricbeat.yml –d “publish”  Windows:  1. PS > scripts\import\_dashboards.exe –es http://<IP>:9200 -user elastic –pass changeme  2. PS > C:\Program Files\Metricbeat> Start-Service metricbeat  Full instructions on starting Metricbeat are available here: [www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-starting.html](http://www.elastic.co/guide/en/beats/metricbeat/current/metricbeat-starting.html).  Finally, let’s verify that a new metricbeat index has been created and populated with data. Go back to Dev Tools and run this query (3.1):  GET metricbeat-\*/\_search  {  "query": {  "match\_all": {}  }  }  You should see several results in the right pane of the console. If not, please ask a presenter for help. |
| 2. | Open a browser to the IP address of your VM at port 9200.  For example: <http://54.204.162.23:9200>.  We are going to open the dashboard named “**Metricbeat-processes”**.  MetricbeatProcess.png  That should bring up a dashboard page. Find the graph titled “CPU usage per process”. It should look something like this:  MetricbeatDashboard.png  You should see your browser’s process show up in that list. For me I see “Google Chrome”.  We can also confirm this by running a query (3.2):  GET metricbeat-\*/\_search  {  "query" : {  "term": {  "system.process.name": {"value": "Google Chrome"}  }  }  }  **NOTE:** For this exercise you can choose any long-running process, but to make it easy I’ve chosen to use my browser’s process. |
| 3. | The first version of the watch will only have the input and a condition defined. This will trigger every 60 seconds and check if our browser process has been logged during the last 2 minutes. The condition is set to only execute when matching records have been found. Create this watch through the following command (step 3.3) (Change **Google Chrome** to the process you chose to alert on):  PUT /\_xpack/watcher/watch/my\_process\_watch  {  "trigger" : {  "schedule" : {"interval" : "60s"}  },  "input" : {  "search" : {  "request" : {  "indices" : ["metricbeat\*"],  "body" : {  "query" : {  "bool" : {  "must" : {  "term": {  "system.process.name": {"value": "**Google Chrome**"}  }  },  "filter": {  "range": {  "@timestamp": { "gt" : "now-2m"}  }  }  }  }  }  }  }  },  "condition" : {  "compare" : {"ctx.payload.hits.total" : { "gt" : 0}}  }  }  You should see:  {  "\_id": "my\_process\_watch",  "\_version": 1,  "created": true  } |
| 4. | Wait a minute for the watch to trigger, and the inspect the result using the command in step 2.3:  GET .watcher-history-\*/\_search  {  "query": {  "match\_all": {}  },  "sort": { "trigger\_event.triggered\_time": "desc"}  } |
| 5. | For this watch we will be indexing a record into a new index instead of logging to file when the condition of the watch is met. Create a new index using the following command (step 3.4):  PUT my\_alert\_index  {  "mappings" : {  "alert" : {  "properties" : {  "alert\_name" : {"type" : "keyword"},  "alert\_text" : {"type" : "text"},  "alert\_time" : {"type" : "date"}  }  }  }  }  This should return:  {  "acknowledged": true,  "shards\_acknowledged": true  } |
| 6. | We will now add a new action to the watch, which will contain a transform that will create the message we will write to the newly created index. The action looks like this:  "actions": {  "index\_payload" : {  "transform": {  "script": "return ['alert\_name': ctx.watch\_id , 'alert\_text': \"Alerted and action taken\" , 'alert\_time': ctx.trigger.triggered\_time] "  },  "index" : {  "index" : "my\_alert\_index",  "doc\_type" : "alert"  }  }  }  We are going to update our existing watch with this extra piece. Execute query 3.5.  The transform script is written in Elasticsearch’s native scripting language **Painless**. It takes a few fields from the watch execution context **ctx** and returns a record with the fields we defined earlier. Ensure that the full transform script is on a single line following the copy and paste of commands into the Console.  It should return something like this:  {  "\_id": "my\_process\_watch",  "\_version": 14,  "created": false  }  Wait a few moments and then check to see that the watch has been executed (3.6):  GET /my\_alert\_index/\_search  {  "query" : {  "term": {  "alert\_name": {"value": "my\_process\_watch"}  }  }  }  You should see something like:  "hits": [  {  "\_index": "my\_alert\_index",  "\_type": "alert",  "\_id": "AV1NjBvV-kIfG1ATlArb",  "\_score": 1,  "\_source": {  "alert\_text": "Alerted and action taken",  "alert\_time": "2017-07-16T22:36:53.916Z",  "alert\_name": "my\_process\_watch"  }  }  ] |
| 7. | Deactivate the watch (step 3.7):  PUT \_xpack/watcher/watch/my\_process\_watch/\_deactivate |
| 8. | Stop metricbeat from running on your laptop. (CTRL-C) |

You are now done with the Watcher/Alerting Lab!

If you are looking for additional examples, check out our examples repository on GitHub:

<https://github.com/elastic/examples>

## Step 4a - look for specific destination or origins in the flight documents. Applications - error messages in the logs

{

"trigger": {

"schedule": {

"interval": "1d"

}

},

"input": {

"search": {

"request": {

"search\_type": "query\_then\_fetch",

"indices": [

"flights-\*"

],

"types": [],

"body": {

"query": {

"bool": {

"must": [

{

"query\_string": {

"query": "DEST:IAH"

}

},

{

"range": {

"@timestamp": {

"gte": "now-1d-2y",

"lte": "now-2y"

}

}

}

]

}

},

"\_source": [

"CARRIER"

],

"sort": [

{

"@timestamp": {

"order": "desc"

}

}

]

}

}

}

},

"condition": {

"compare": {

"ctx.payload.hits.total": {

"gt": 0

}

}

},

"actions": {

"log": {

"logging": {

"level": "info",

"text": "{{ctx.payload.hits.total}} {{ctx.metadata.dest}} flights have occured:{{#ctx.payload.hits.hits}}{{\_source.CARRIER}},{{/ctx.payload.hits.hits}}"

}

}

},

"throttle\_period\_in\_millis": 900000,

"metadata" :{

"dest": "IAH"

}}

## Step 4a’ Use search dsl to get the unique carriers for the documents rather than the hits in order as above.

{

"trigger": {

"schedule": {

"interval": "1d"

}

},

"input": {

"search": {

"request": {

"search\_type": "query\_then\_fetch",

"indices": [

"flights-\*"

],

"types": [],

"body": {

“size”:1000,

"query": {

"bool": {

"must": [

{

"query\_string": {

"query": "ORIGIN:{{ctx.metadata.dest}}"

}

},

{

"range": {

"@timestamp": {

"gte": "now-1d-2y",

"lte": "now-2y"

}

}

}

]

}},

"collapse": {

"field": "CARRIER"

},

"\_source": [

"CARRIER"

],

"sort": [

{

"@timestamp": {

"order": "desc"

}

}

]

}

}

}

},

"condition": {

"compare": {

"ctx.payload.hits.total": {

"gt": 0

}

}

},

"actions": {

"log": {

"logging": {

"level": "info",

"text": "{{ctx.payload.hits.total}} {{ctx.metadata.dest}} flights have occured:{{#ctx.payload.hits.hits}}{{\_source.CARRIER}},{{/ctx.payload.hits.hits}}"

}

}

},

"metadata": {

"dest": "IAH"

},

"throttle\_period\_in\_millis": 900000

}

## Step 4b:

{

"metadata": {

"threshold": 180,

"window\_period": "1d"

},

"trigger": {

"schedule": {

"interval": "1d"

}

},

"input": {

"search": {

"request": {

"indices": [

"flights-\*"

],

"body": {

"aggs": {

"carrier": {

"terms": {

"field": "CARRIER"

},

"aggs": {

"delay": {

"max": {

"field": "ARR\_DELAY"

}

}

}

}

},

"query": {

"bool": {

"filter": [

{

"range": {

"@timestamp": {

"gte": "now-{{ctx.metadata.window\_period}}-2y",

"lte": "now-2y"

}

}

},

{

"range": {

"ARR\_DELAY": {

"gte": "{{ctx.metadata.threshold}}"

}

}

}

]

}

}

}

}

}

},

"condition": {

"compare": {

"ctx.payload.hits.total": {

"gt": 0

}

}

},

"actions": {

"log": {

"logging": {

"text": {

"inline": "Some carriers are over {{ctx.payload.threshold}}:{{#ctx.payload.aggregations.carrier.buckets}}{{key}}{{.}}:{{/ctx.payload.aggregations.carrier.buckets}}"

}

}

}

}

}

## Step 4b: OPTIONAL: Detecting missing “things” with Alerting

This exercise will alert on missing “things” that you configure in the watch in the flight data set over the time windows that you select.

The watch is designed to be a reusable solution for comparing two lists that are generated from two different time buckets, in this flight data we can detect missing destination or origin airports, carriers and states could also be used for example. In a generic security sense this can be used to look at things like new systems sending logs or systems that sent logs yesterday but are no longer sending logs now.

|  |  |
| --- | --- |
| 1. | Open Management -> Index Patterns in Kibana and if not already present create a new index pattern flights-\*.  Go to Discover and get familiar with the structure of the records that are logged. |
| 2. | The watch we are creating is set to run every 1 hour and look at the data for the last 2 days but can be changed in the metadata and used interactively with the \_execute endpoint.  The full watch definition together with a condition is available in the Console under step 4.1. We will now go through it step by step before registering it. The first part specifies the window period and the last\_period, an offset is also included to look at historical data as if it was in real time as metadata and defined how frequently the watch will execute.  "metadata": {  "window\_period": "48h",  "last\_period": "24h",  "thing": "DEST",  "timefield": "@timestamp",  "offset": "2y"  },  "trigger": {  "schedule": {  "interval": "1h"  }  },  The input section of the watch consists of a search that breaks the data into two date histograms and by thing. The first one is named history and runs an aggregation that returns the number of “things” events buckets in the now-offset-window\_period to now-offset-last\_period, the second bucket is now-offset to now-offset-last\_period:  "input": {  "search": {  "request": {  "indices": "flights-\*",  "body": {  "query": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.window\_period}}",  "lte": "now-{{ctx.metadata.offset}}"  }  }  },  "aggs": {  "periods": {  "filters": {  "filters": {  "history": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.window\_period}}",  "lte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.last\_period}}"  }  }  },  "last\_period": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.last\_period}}",  "lte": "now-{{ctx.metadata.offset}}"  }  }  }  }  },  "aggs": {  "things": {  "terms": {  "field": "{{ctx.metadata.thing}}",  "size": 10000  }  }  }  }  },  "size": 0  }  }  }  },  The condition is simple, as we want to alert whenever the history has more buckets of data than the last period:  "condition": {  "script": {  "inline": "return ctx.payload.aggregations.periods.buckets.history.things.buckets.size() > ctx.payload.aggregations.periods.buckets.last\_period.things.buckets.size();",  "lang": "painless"  }  },  The script checks for the actual values that are missing from the last\_period set of buckets by comparing to the history buckets in the transform section:  "actions": {  "log": {  "transform": {  "script": {  "inline": "def last\_period=ctx.payload.aggregations.periods.buckets.last\_period.things.buckets.stream().map(e -> e.key).collect(Collectors.toList()); return ctx.payload.aggregations.periods.buckets.history.things.buckets.stream().map(e -> e.key).filter(p -> !last\_period.contains(p)).collect(Collectors.toList());",  "lang": "painless"  }  }, |
| 3. | Use the Watcher activity in the Management section of Kibana to add the watch console under step 4.1.  {  "metadata": {  "window\_period": "96h",  "last\_period": "72h",  "thing": "DEST",  "timefield": "@timestamp",  "offset": "2y"  },  "trigger": {  "schedule": {  "interval": "1h"  }  },  "input": {  "search": {  "request": {  "indices": "flights-\*",  "body": {  "query": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.window\_period}}",  "lte": "now-{{ctx.metadata.offset}}"  }  }  },  "aggs": {  "periods": {  "filters": {  "filters": {  "history": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.window\_period}}",  "lte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.last\_period}}"  }  }  },  "last\_period": {  "range": {  "{{ctx.metadata.timefield}}": {  "gte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.last\_period}}",  "lte": "now-{{ctx.metadata.offset}}-{{ctx.metadata.last\_period}}+24h"  }  }  }  }  },  "aggs": {  "things": {  "terms": {  "field": "{{ctx.metadata.thing}}",  "size": 10000  }  }  }  }  },  "size": 0  }  }  }  },  "condition": {  "script": {  "inline": "return ctx.payload.aggregations.periods.buckets.history.things.buckets.size() > ctx.payload.aggregations.periods.buckets.last\_period.things.buckets.size();",  "lang": "painless"  }  },  "actions": {  "log": {  "transform": {  "script": {  "inline": "def last\_period=ctx.payload.aggregations.periods.buckets.last\_period.things.buckets.stream().map(e -> e.key).collect(Collectors.toList()); return ctx.payload.aggregations.periods.buckets.history.things.buckets.stream().map(e -> e.key).filter(p -> !last\_period.contains(p)).collect(Collectors.toList());",  "lang": "painless"  }  },  "logging": {  "text": "{{ctx.metadata.thing}} missing in the last {{ctx.metadata.last\_period}} that were present in the prior day: {{#ctx.payload.\_value}}{{.}} {{/ctx.payload.\_value}}"  }  }  }  } |
| 4. | Simulate the watch using the second tab of the Watcher UI to see if the windows of time you configured is missing destinations, you might need to update the metadata for the windows of time to search as the offset and/or thing to see results at the bottom of the output from simulation.  Example log output at the bottom of the results showing 3 destinations present in the prior window but not the current window of time:  "logging": {  "logged\_text": "DEST missing in the last 24h that were present in the prior day: IAG ADK UST "  } |
| 5. | Experiment with changing the windows and things to find your own interesting results in the dataset. |