ExtraHop Open Data Stream for ELK

ExtraHop, in conjuncting with Elasticsearch, Logstash, and Kibana (ELK) can be used to drive greater operational insight by combining a broader data set to facilitate troubleshooting, optimization, and business decision making.

This guide shows how to configure both the ExtraHop system and the ELK stack to stream wire data from an ExtraHop appliance to the ELK system. There are two main methods of streaming data from the ExtraHop platform to ELK:

- Use Open Data Stream (ODS) for HTTP to insert data directly into Elasticsearch
- Stream data with Open Data Stream (ODS) for Syslog to Logstash where it is filtered and parsed into an Elasticsearch-compatible format

Writing directly to the Elasticsearch datastore typically performs better than sending logs to Logstash and requires less configuration. To write directly to Elasticsearch, you must use ExtraHop firmware version 4.1 and have overhead for more trigger load on the ExtraHop system.

The examples in this guide show how to get a basic solution working. Because both platforms are highly extensible, there is room to expand based on each unique environment.

This guide assumes that you already have the ELK stack deployed and functioning. For assistance customizing and tuning the ELK environment, refer to the Elasticsearch documentation at http://www.elastic.co/guide/.

Integration using ODS for HTTP

Writing directly to the Elasticsearch datastore takes advantage of the ExtraHop ODS for HTTP and Elasticsearch's REST API. The API's endpoint is set to port 9200 by default but can modified in the configuration file at */etc/elasticsearch/elasticsearch.yml*. If the ELK system is using non-default configurations, make suitable changes to the ExtraHop configurations shown below.

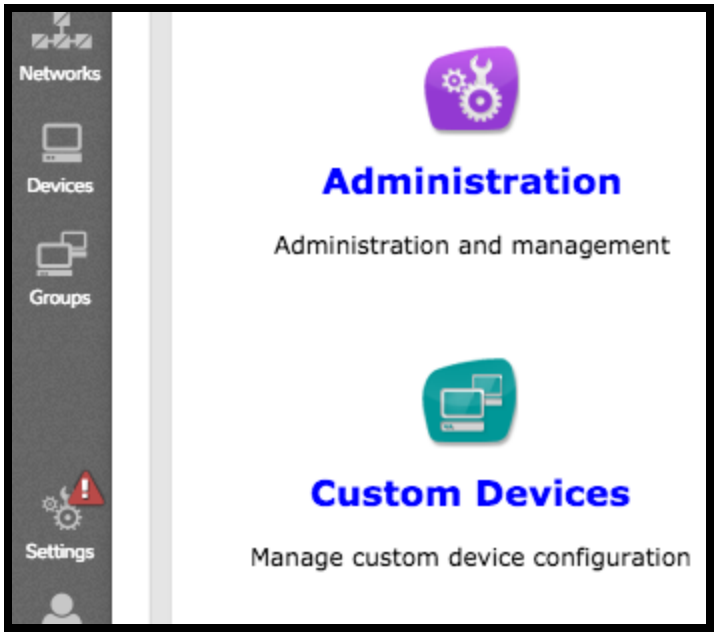
Configuring Open Data Stream is a two-step process. First, set the HTTP destination. Second, configure Application Inspection Triggers (triggers) to send the desired data.

Set the HTTP Destination

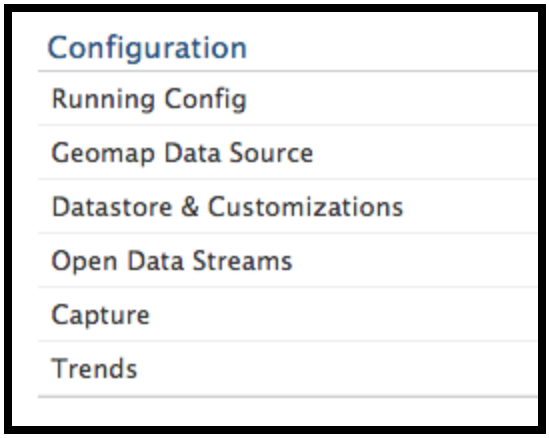
The HTTP destination is configured in the Administration UI of the ExtraHop system (`https://<extrahop_ip>/admin`).

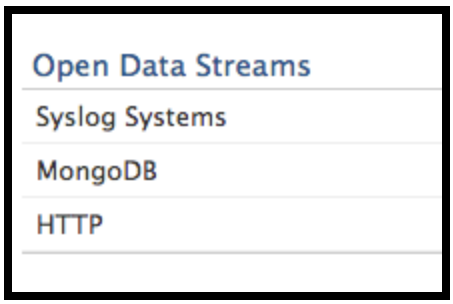
To configure the ExtraHop system:

1. To access the Administration UI, click **Settings** in the left navigation bar, and then click **Administration**.



2. Go to the Configuration section, click **Open Data Streams**, and then click **HTTP**.





3. After the last DataStream Configuration, click **Add New**.

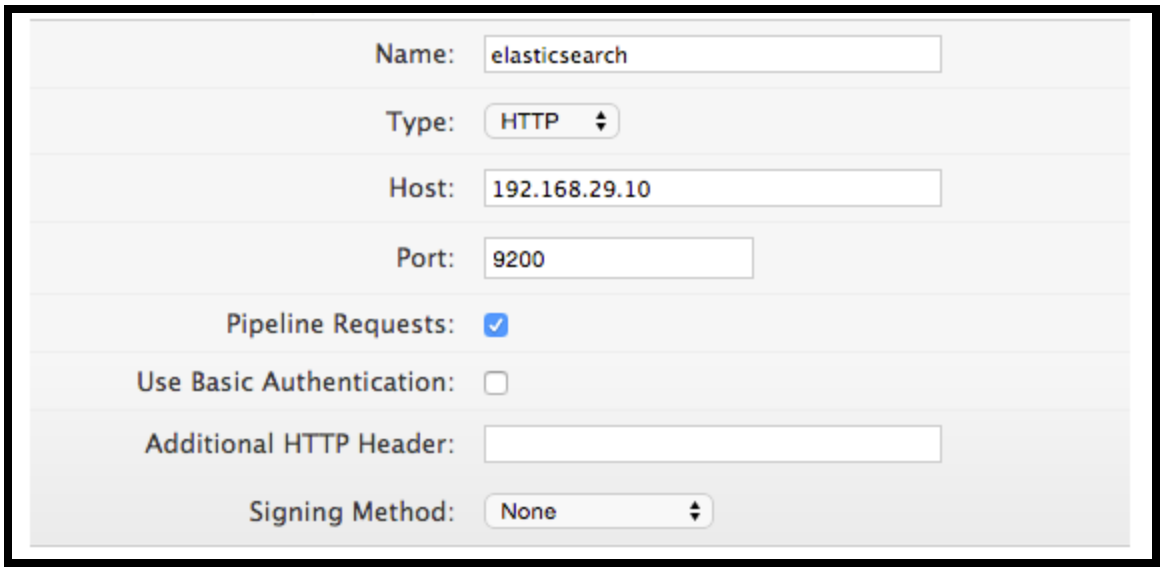
4. In the **Name** field, enter the name for this HTTP stream. The name will be used in the trigger, so a descriptive name such as "elasticsearch" is useful.

5. Leave the **Type** field set to HTTP.

6. In the **Host** field, enter the hostname or IP address of the system with ELK installed.

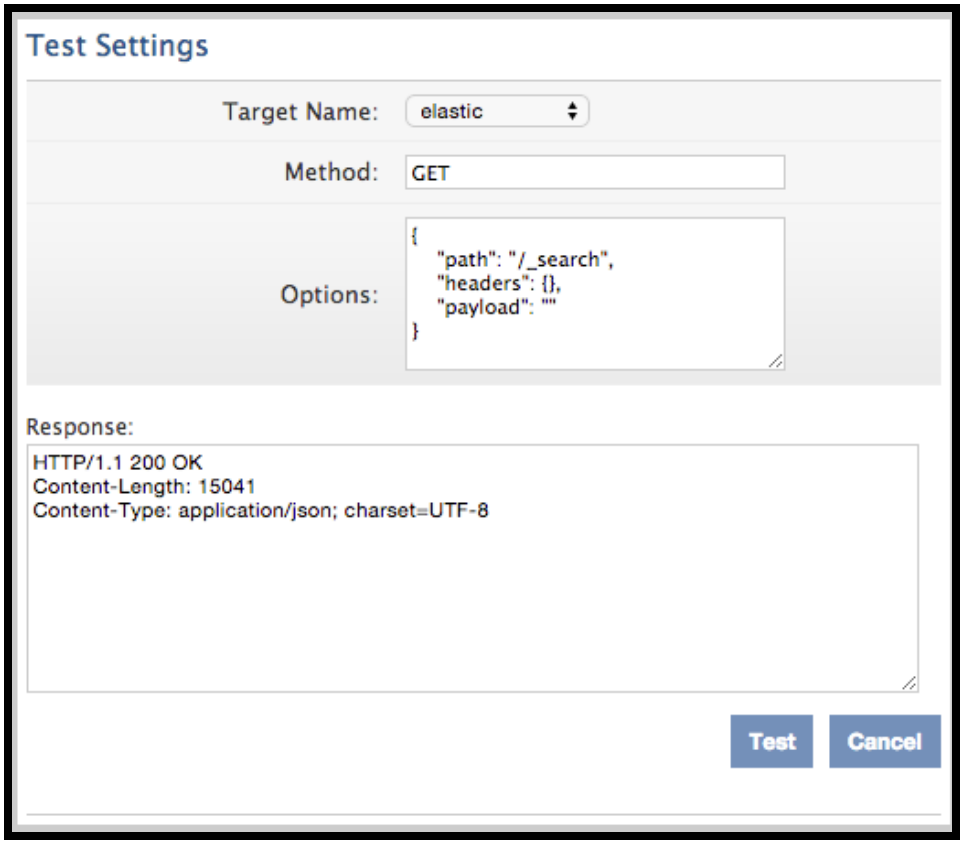
7. In the **Port** field, set the port to **9200,** unless Elasticsearch has been configured to use a different port.

8. Accept the rest of the defaults, and save the configuration.



To test the configuration, click **Test Settings**. In the **Options**, copy the following code or change the path to "/_search". Click **Test**, and the **Response** field should come back with an HTTP 200 OK.

```
{
    "path": "/_search",
    "headers": {},
    "payload": ""
}
```



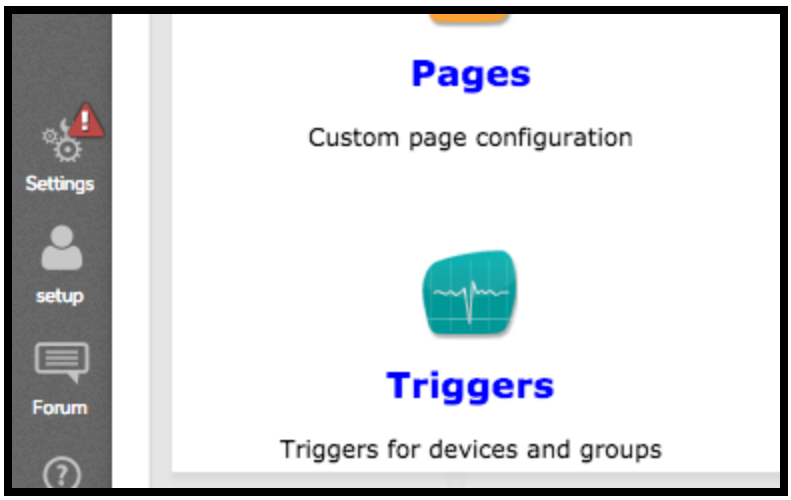
The next step is to configure triggers to send data to the ELK endpoint

Configure Application Inspection Triggers

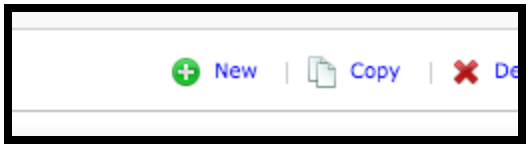
Once the ExtraHop system has been configured with an HTTP endpoint, create one or more triggers. The triggers supply a policy via JavaScript code that covers how to transform wire data into a JSON document suitable for inserting into Elasticsearch. Appendix A at the end of this guide contains several sample triggers that you can copy into the **Trigger Editor** in the ExtraHop Web UI and customize for your needs.

To create the trigger:

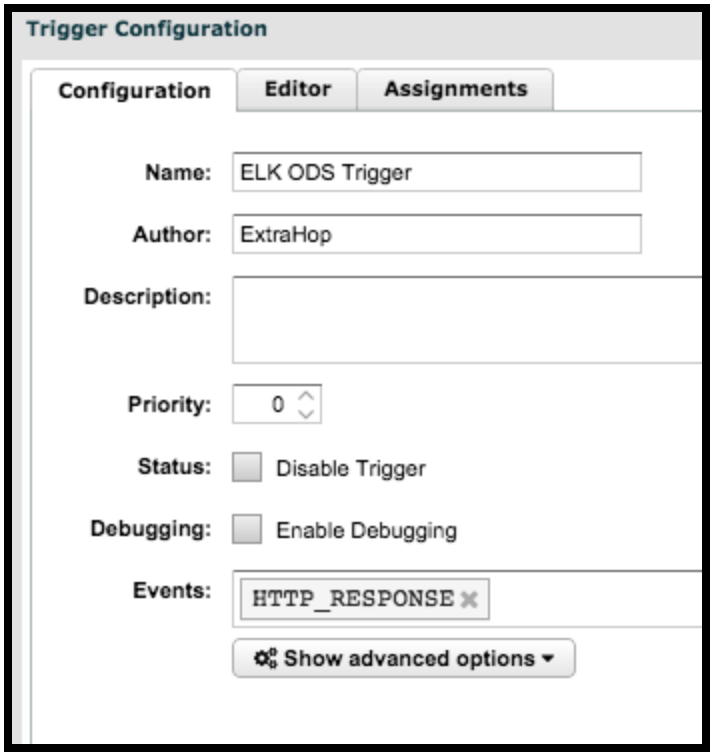
1. Click **Settings** in the navigation bar, and then click **Triggers**.



2. Click **New** at the top of the dialog. An additional dialog appears.



Enter a name for the trigger and select at least one event with which the trigger should be associated. The events that you choose depend upon the type of data that you are sending. This example is extracting HTTP data, and the HTTP_RESPONSE event is selected.



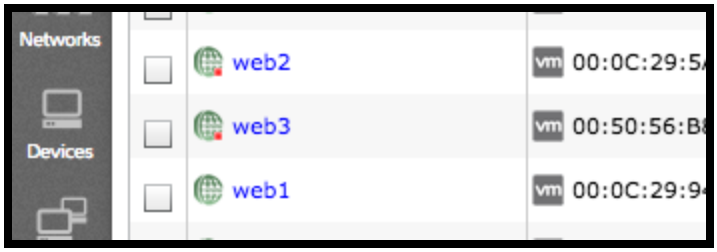
4. Once you have chosen an event, click the **Editor** tab to enter your trigger code. This pane lets you write JavaScript that you can use to build a data structure suitable for sending to Elasticsearch. To send data to Elasticsearch, you must specify the JSON text to send. The code will look similar to the following example:

```
var date = new Date();
var payload = {
    'ts'         : date.toISOString(), // Timestamp recognized by
Elasticsearch
    'eh_event'   : 'http',
    'my_path'    : HTTP.path};

var obj = {
    'path'     : '/extrahop/http', // Add to extrahop index
    'headers'  : {},
    'payload'  : JSON.stringify(payload)} ;

Remote.HTTP('elasticsearch').request('POST', obj);
```

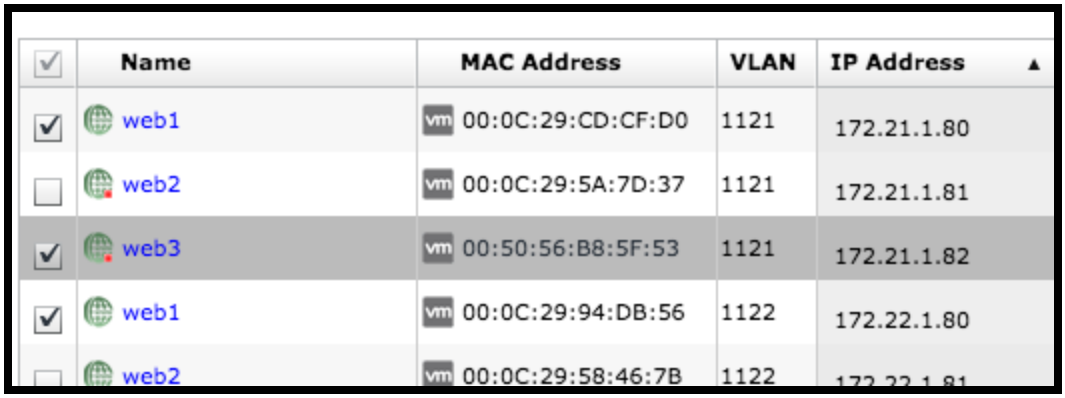
5. Click the **Save** button.

Once a trigger is created and associated with events, it must be assigned to a set of devices. In a production environment, we recommend that you assign the trigger to only the specific devices with data of interest. To assign the trigger to specific devices:

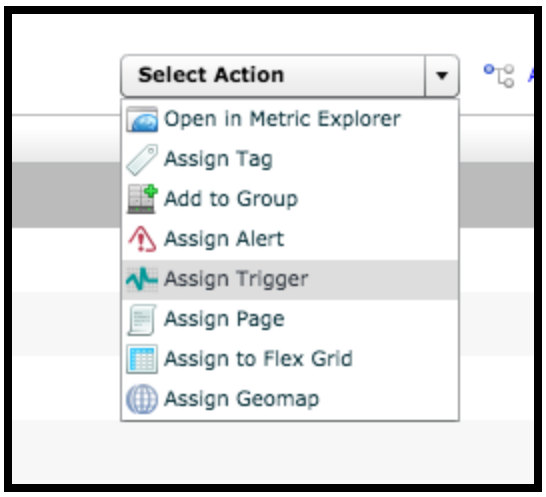
1. Click **Devices** in the navigation bar.



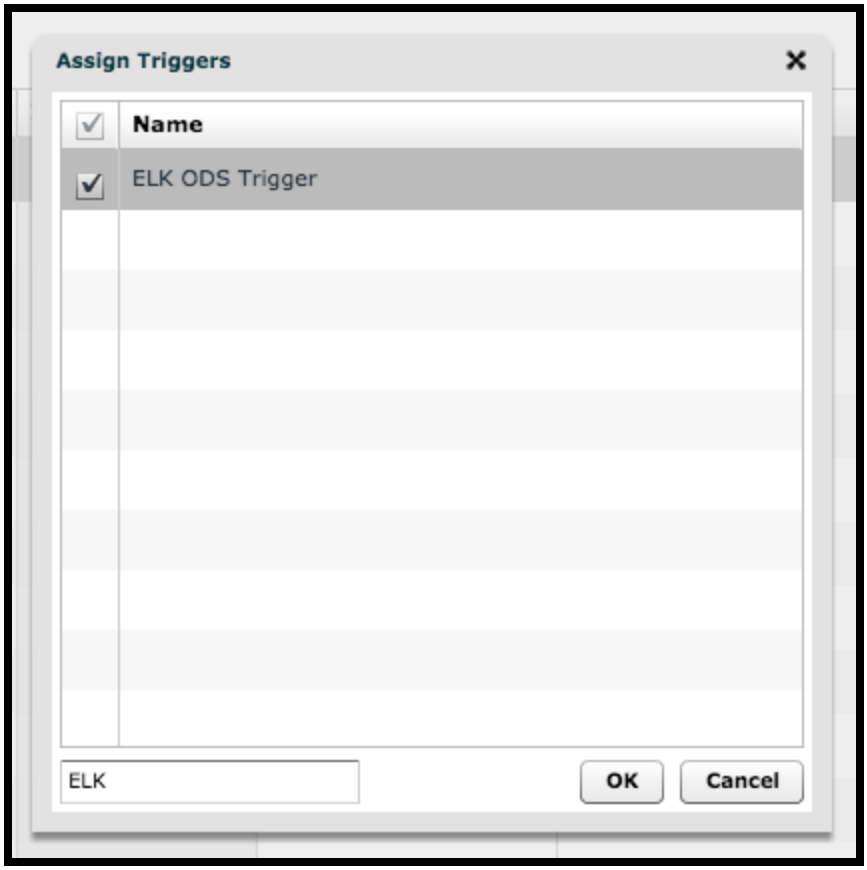
2. Select the devices to associate with the trigger.



3. Click the **Select Action** dropdown list, and select **Assign Trigger**.



4. In the **Assign Triggers** dialog box, select the checkbox next to the trigger, and click **OK**.



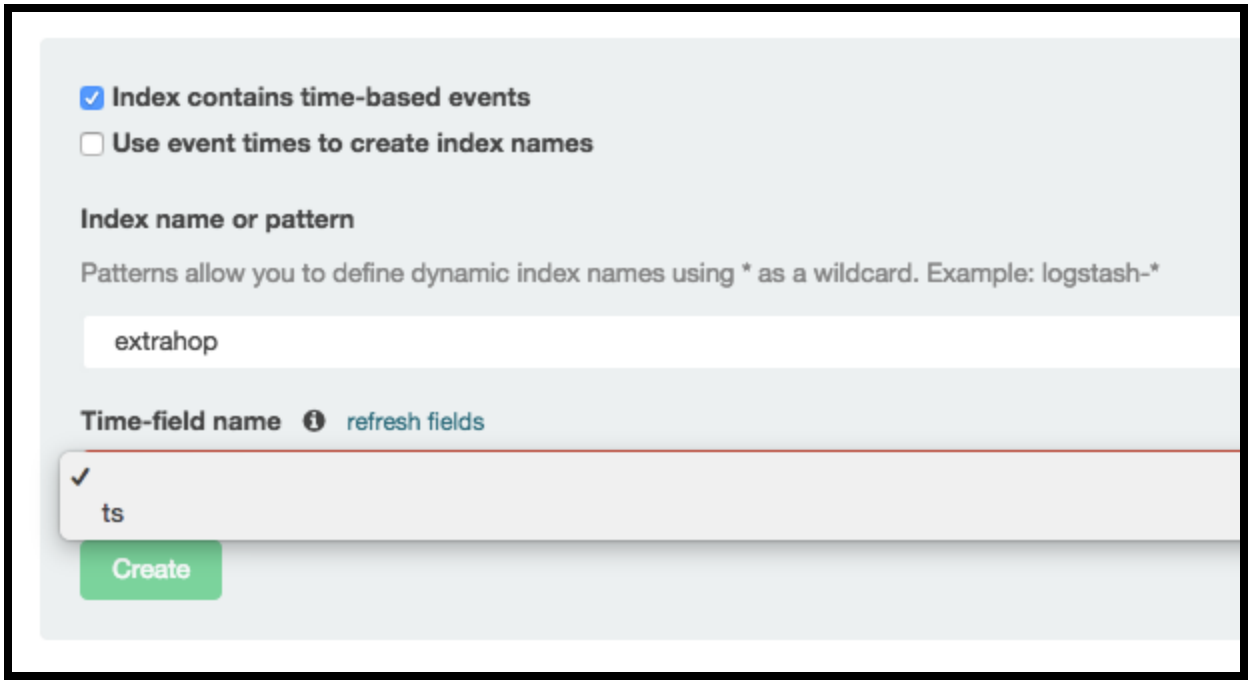
This example trigger creates a simple JavaScript object (payload) with three properties (a timestamp, "eh_event", which identifies the ExtraHop event, and "my_path", which contains the HTTP path). The argument for the Remote.HTTP() function call needs to match the name of the ODS for HTTP endpoint that was configured in the **Setting HTTP Destinations** section above. The payload is then added to the *extrahop* index with a document type of "http".

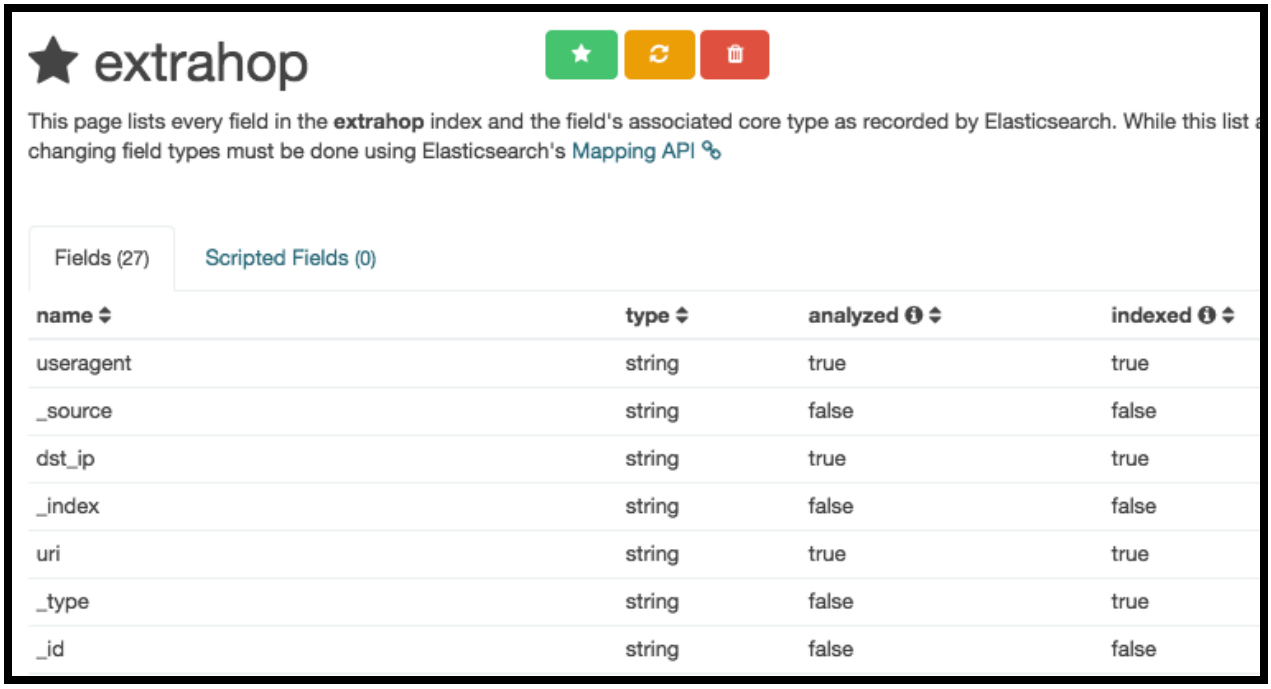
This is an intentionally simple example, but those with some experience writing triggers already know that the object that is inserted can be dynamically created at runtime, freely structured by the trigger author, and made up of any wire data available via the Trigger API which is documented in the *ExtraHop Trigger API* document on the ExtraHop Customer Forum.

For more information about creating triggers, refer to the ExtraHop website. Also, refer to Appendix A at the end of this guide for more full-featured examples using ODS for HTTP.

Configure an Index Pattern

The last step before using your data is to configure an index pattern in the Kibana UI, which is modified under **Settings** > **Indices**. Select the appropriate timestamp field, or if no field is supplied, uncheck the "Index contains time-based events" check box. Once configured, the metrics fields will automatically be identified by Elasticsearch.





Integration using ODS for Syslog

When using ExtraHop's ODS for Syslog to integrate the two platforms, Logstash must first be configured to accept and parse data inputs correctly before configuring ExtraHop to send data. The following gives step-by-step instructions for configuring both.

Elasticsearch, Logstash, and Kibana (ELK)

Configure Logstash Filters

This section covers configuring Logstash to consume the ExtraHop ODS syslog. Logstash requires three sections to be present in order to consume the syslog data: the input, the filter, and the output. There are many ways to configure Logstash to accept data via remote syslog and insert it into Elasticsearch, but the easiest way to get data from an ExtraHop appliance is to accept data in JSON format.

Input: To configure Logstash to accept syslog data from ExtraHop, create or edit `/etc/logstash/conf.d/0_inputs.conf` with a text editor (such as vi) and enter the following text:

```
input {
  tcp {
    port => 1514
    type => syslog
  }
  udp {
    port => 1514
    type => syslog
  }
}
```

This configuration instructs Logstash to listen for syslog data on port 1514, sent using either TCP or UDP. Note that the ExtraHop system will have to be configured to use the same port for its syslog destination, which is covered below.

Filter: To configure Logstash to understand the JSON input, create or edit `/etc/logstash/conf.d/10_extrahop_filter.conf` and enter the following text:

```
filter {
  if [type] == "syslog" and [message] =~ /eh_event/ {
    grok {
      match => { "message" => \
"<%{POSINT:syslog_pri}>%{TIMESTAMP_ISO8601:syslog_timestamp} %{HOSTNAME:hostname} \
%{GREEDYDATA:json_message}" }
    remove_field => ["message"]
    }
     json {
# parse JSON from "json_message" extracted under "match",
# put resulting structure in "data" field
      source => "json_message"
      remove_field => ["json_message"]
    }
    date {
      timezone => "America/Los_Angeles"
      match => [ "syslog_timestamp", "ISO8601" ]
      locale => "en"
   }
  }
}
```

The filter first checks to see if the message is a syslog message and whether it contains the string "eh_event" in the message. If the message meets that initial test, the filter identifies the syslog PRI, timestamp, and message body, which are then parsed individually. The message body format is a modifiable JSON blob. For more information, refer to the **Configure Application Inspection Triggers** section.

Output: Finally, to configure the Logstash output method, create or edit `/etc/logstash/conf.d/30_outputs.conf`, and enter the following text:

```
output {
  if [type] == "syslog" and "_grokparsefailure" in [tags] {
    file { path => "/var/log/logstash/failed_syslog_events-%{+YYYY-MM-dd}" }
  } else {
      elasticsearch_http {
        host => localhost
      }
  }
  stdout {
    codec => rubydebug }
}
```

The output is configured to push successfully parsed messages into Elasticsearch. Unsuccessfully parsed messages are logged on the Logstash server for future analysis. You must restart the Logstash process to use this new configuration.

Configure the ExtraHop System

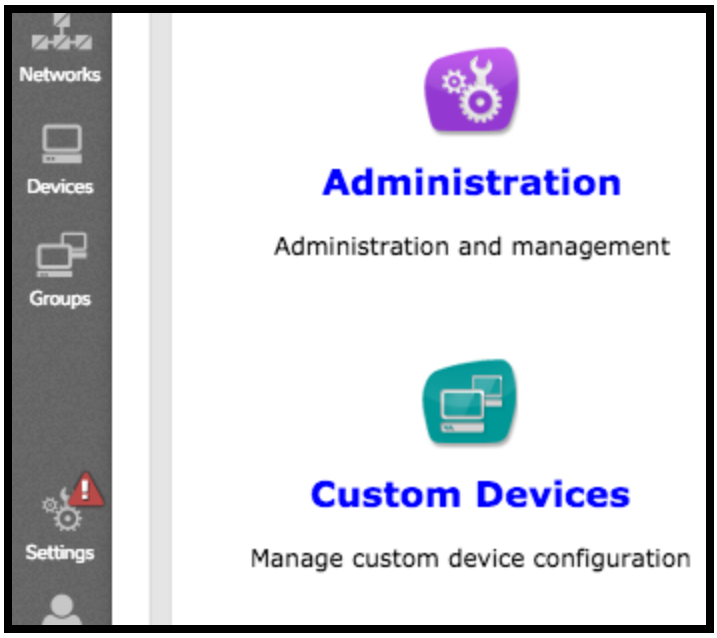
You can send wire data into Elasticsearch via Logstash using ODS over syslog. Configuring ODS is a two-step process. First, set the syslog destination. Second, configure triggers to send the desired data.

Set the Syslog Destination

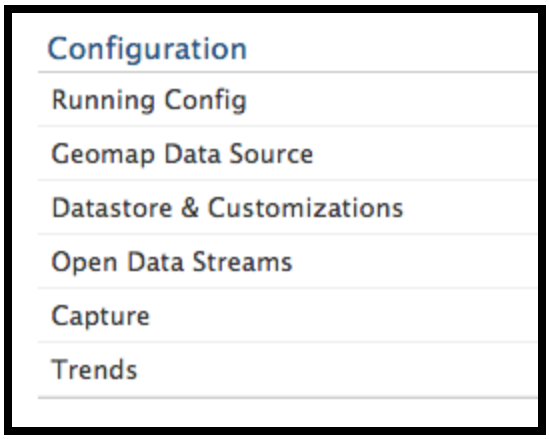
The syslog destination is configured in the Administration UI of the ExtraHop system (`https://<extrahop_ip>/admin`).

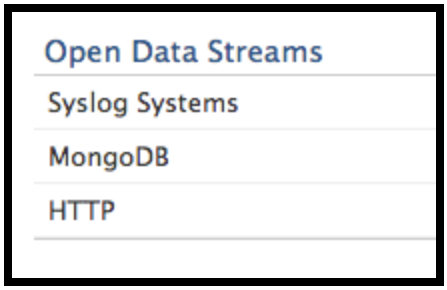
To configure the ExtraHop system:

1. To access the Administration UI, click **Settings** in the left navigation bar, and then click **Administration**.

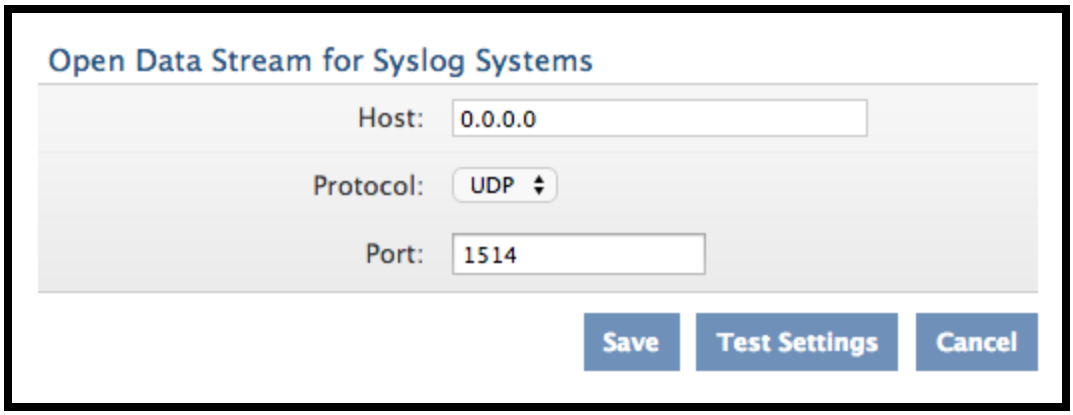


2. Go to the Configuration section, click **Open Data Streams**, and then click **Syslog Systems**.





3. In the **Host** field, enter the hostname or IP address of the system with ELK installed.
4. Leave the **Protocol** field set to **UDP**.
5. Set the **Port** to **1514** to correspond with the Logstash JSON input.

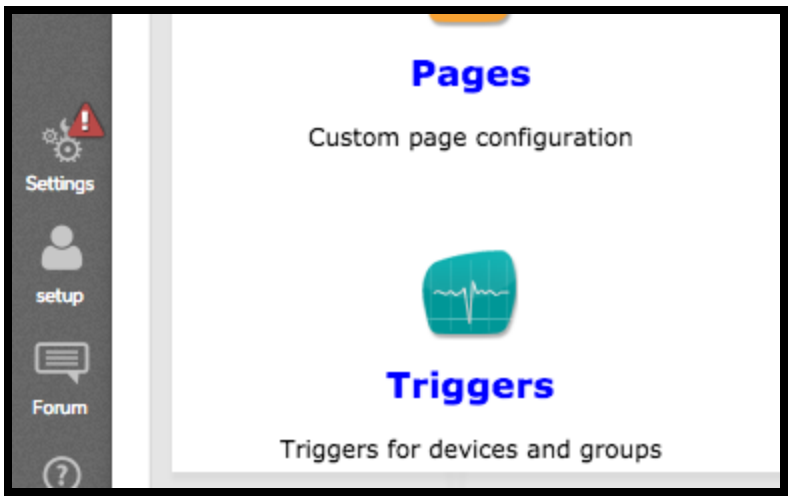


Configure Triggers

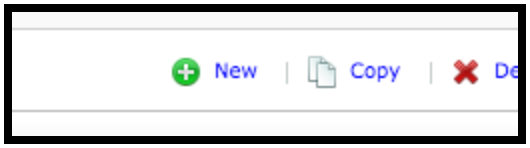
Once the ExtraHop system has been configured with a syslog endpoint, create one or more triggers. These triggers supply a policy via JavaScript code that covers how to transform wire data into a JSON document suitable for inserting into Elasticsearch via Logstash. The Appendix B at the end of this guide contains several trigger samples that you can copy into the **Trigger Editor** in the ExtraHop Web UI and customize for your needs.

To create the Application Inspection Trigger:

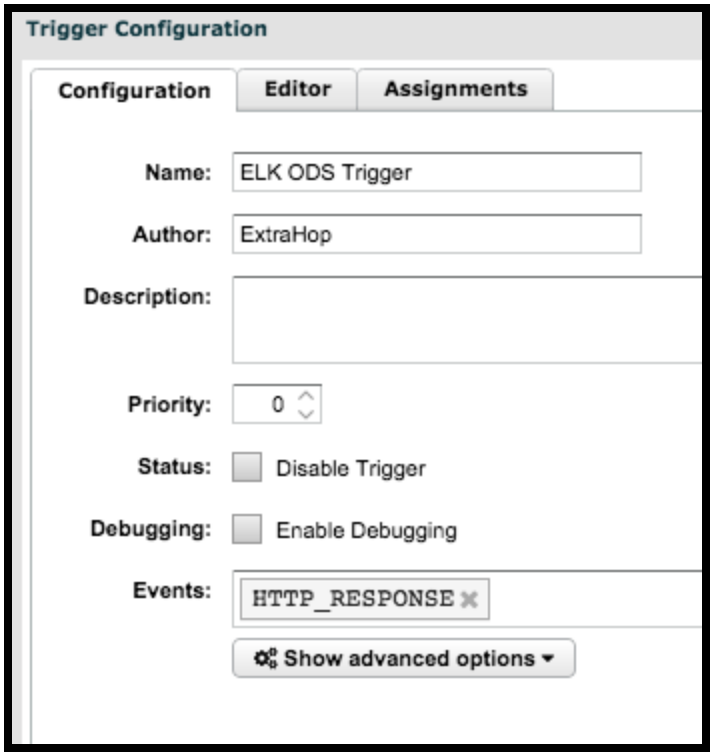
1. Click **Settings** in the navigation bar, and then click **Triggers**.



2. Click **New** at the top of the dialog. An additional dialog appears.



3. Enter a name for the trigger, and select at least one event with which the trigger should be associated.The events that you choose depend upon the type of data that you are sending. This example is extracting HTTP data, and the HTTP_RESPONSE event is selected.



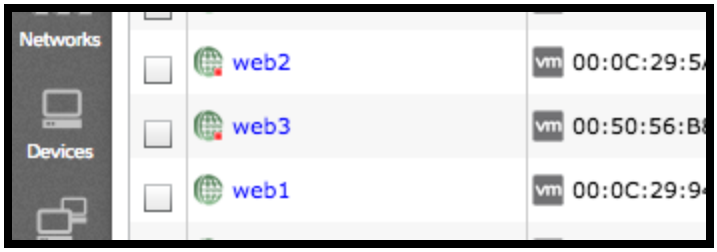
4. Once you have chosen an event, click the **Editor** tab to enter your trigger code. This pane lets you write JavaScript that you can use to build a data structure suitable for sending to Elasticsearch via Logstash. To send data to Logstash, you must specify the JSON text to send. The code will look similar to the following example:

```
var my_object = {
    'eh_event' : 'http',
    'my_path' : HTTP.path,
};

RemoteSyslog.info(JSON.stringify(my_object));
```

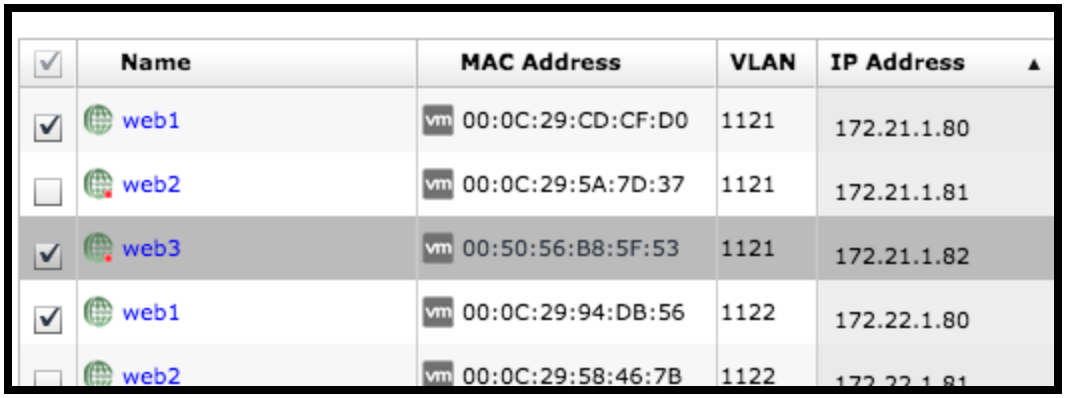
5. Click the **Save** button.

Once a trigger is created and associated with an event, it must be assigned to a set of devices. In a production environment, we recommend that you assign the trigger to only the specific devices with data of interest. To assign the trigger to specific devices:

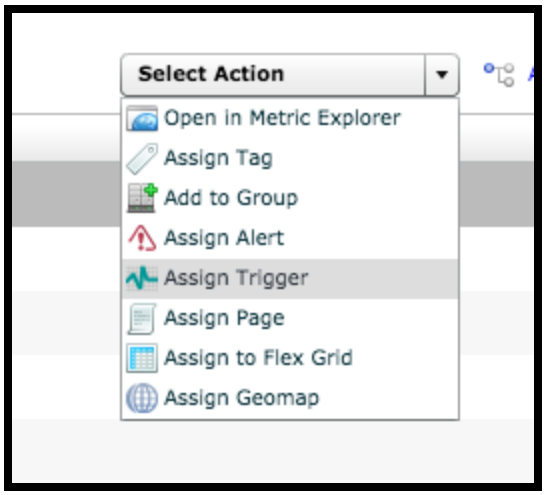
1. Click **Devices** in the navigation bar.



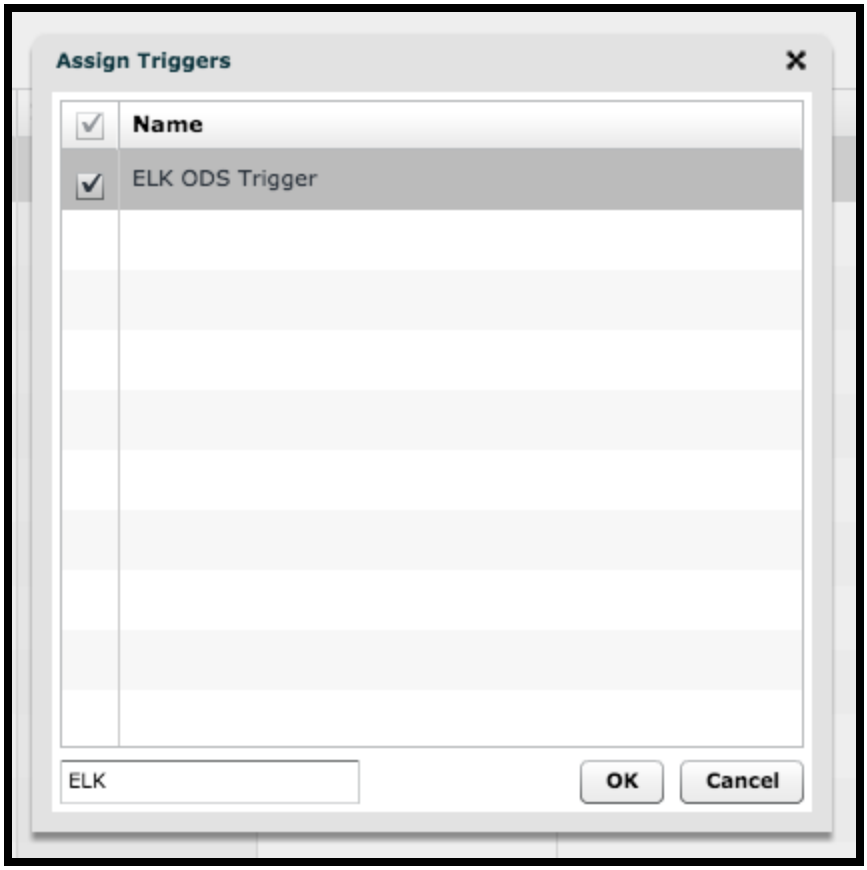
2. Select the devices to associate with the trigger.



3. Click the **Select Action** dropdown list, and select **Assign Trigger**.



4. In the **Assign Triggers** dialog box, select the checkbox next to the trigger, and click **OK**.

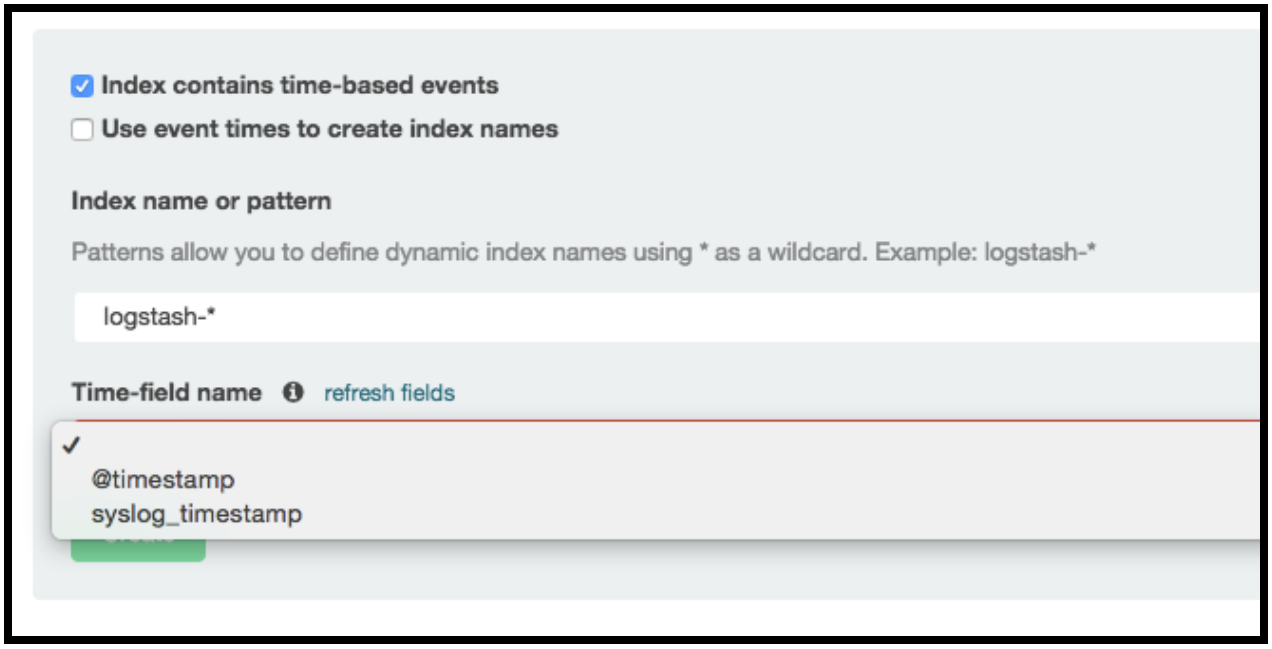


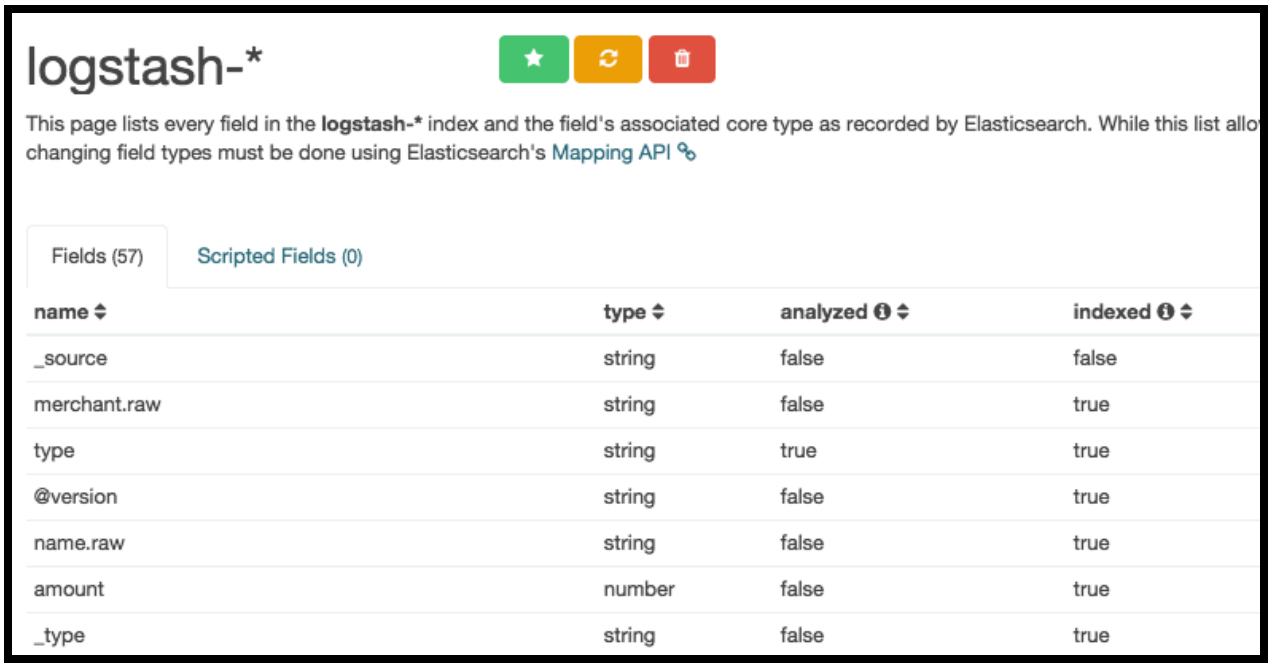
This example trigger creates a simple JavaScript object (my_object) with two properties ("eh_event", which is required for the Logstash filter, and "my_path", which contains the HTTP path) before inserting it into Elasticsearch via Logstash. This is an intentionally simple example, but those with some experience writing triggers already know the object that is inserted can be dynamically created at runtime, freely structured by the trigger author, and made up of any wire data available via the Trigger API which is documented in the *ExtraHop Trigger API* document on the ExtraHop Customer Forum.

For more information about creating triggers, refer to the ExtraHop website. Also, refer to Appendix B at the end of this guide for more full-featured examples using ODS for Syslog.

Configure an Index Pattern

The last step before using your data is to configure an index pattern in the Kibana UI, which is modified under **Settings** > **Indices**. Select the appropriate timestamp field, or if no field is supplied, uncheck the "Index contains time-based events" check box. Once configured, the metrics fields will automatically be identified by Elasticsearch.





Leverage Your Data

The ExtraHop ODS capabilities send wire data from an ExtraHop appliance to a flexible datastore for subsequent multi-variable, post-hoc analysis. Combining the ELK stack and ODS gives customers control over their own data, the ability to combine it with other sources of data, and the flexibility to use a variety of tools to analyze the data and extract value from it. Furthermore, because the wire data that is exported comes from a user-provided trigger, the choice of what data to export, how it is formed, and how often to send it is in the user's control. This offers maximum choice, flexibility, and openness to organizations using the ExtraHop platform.

Appendix A: Default ODS for HTTP Triggers

The following triggers are an example and starting point for building out an ODS for HTTP integration. Fields can be added or removed based on need.

Note: The following triggers assume the Elasticsearch destination has been named "elasticsearch". If this is not the case, then the argument for the Remote.HTTP() function needs to be changed to match.

HTTP

```
// Event: HTTP_REQUEST
// Avoid snake-eating-tail monitoring situation
if (HTTP.userAgent.indexOf('ExtraHop') > -1) {
    return;
}

var date = new Date();

var payload = {
    'ts'           : date.toISOString(),
    'eh_event'     : 'http',
    'src_ip'       : Flow.client.ipaddr.toString(),
    'dest_ip'      : Flow.server.ipaddr.toString(),
    'uri'          : HTTP.uri,
    'query'        : HTTP.query };

var obj = {
    'path'     : '/extrahop/http',
    'headers'  : {},
    'payload'  : JSON.stringify(payload)} ;

Remote.HTTP('elasticsearch').request('POST', obj);
```

DNS

```
// Event: DNS_RESPONSE
var date = new Date();

var payload = {
    'ts'           : date.toISOString(),
    'eh_event'     : 'dns',
    'src_ip'       : Flow.client.ipaddr.toString(),
    'dest_ip'      : Flow.server.ipaddr.toString(),
    'qname'        : DNS.qname,
    'qtype'        : DNS.qtype,
    'opcode'       : DNS.opcode,
    'tprocess'     : DNS.tprocess
};
// If you want every answer, build loop here
var answer = DNS.answers[0];

if (answer !== undefined) {
```

```
        payload.ans_name = answer.name;
        payload.ans_ttl = answer.ttl;
        payload.ans_type = answer.type;

    if (answer.data !== null) {
        payload.ans_data = answer.data;
    }
}

if (DNS.error !== null) {
    payload.error = DNS.error;
}
var obj = {
    'path'     : '/extrahop/dns',
    'headers'  : {},
    'payload'  : JSON.stringify(payload)} ;

Remote.HTTP('elasticsearch').request('POST', obj);
```

Database

```
// Event: DB_REQUEST, DB_RESPONSE
// Set to true for full SQL statements
var SHOW_FULL_STATEMENT = false;

if (event == 'DB_REQUEST' && SHOW_FULL_STATEMENT) {
    Flow.store.db_statement = DB.statement || DB.procedure;
}

else if (event == 'DB_RESPONSE') {
    var date = new Date();

    var payload = {
        'ts'          : date.toISOString(),
        'eh_event'    : 'database',
        'method'      : DB.method,
        'user'        : DB.user,
        'req_L2bytes' : DB.reqL2Bytes,
        'rsp_L2bytes' : DB.rspL2Bytes
    }
    if (Flow.store.db_statement !== null && SHOW_FULL_STATEMENT) {
        payload.query = Flow.store.db_statement;
        Flow.store.db_statement = null;
    }

    if (DB.error != null) {
        payload.error = DB.error;
    }

    if (DB.tprocess) {
        payload.tprocess = DB.tprocess;
```

```
    }

    var obj = {
        'path'     : '/extrahop/database',
        'headers'  : {},
        'payload'  : JSON.stringify(payload)} ;

    Remote.HTTP('elasticsearch').request('POST', obj);
}
```

Storage

```
// Event: CIFS_RESPONSE
var date = new Date();

var payload = {
    "ts"           : date.toISOString(),
    "eh_event"     : "cifs",
    "method"       : CIFS.method,
    "user"         : CIFS.user,
    "access_time"  : CIFS.accessTime,
    "req_L2bytes"  : CIFS.reqL2Bytes,
    "rsp_L2bytes"  : CIFS.rspL2Bytes };

if (CIFS.resource != null) {
    payload.filename = CIFS.resource;
}
if (CIFS.error != null) {
    payload.error= CIFS.error;
}


var obj = {
    'path'     : '/extrahop/storage',
    'headers'  : {},
    'payload'  : JSON.stringify(payload)} ;

Remote.HTTP('elasticsearch').request('POST', obj);
```

Appendix B: Default ODS for Syslog Triggers

The following triggers are an example and starting point for building out an ODS for syslog integration. Fields can be added or removed based on need.

Note: According to RFC 3164 (section 4.1), a syslog message must be 1024 bytes or less, but extended syslog with TCP can support larger message sizes. Messages will be truncated if the message size exceeds the the supported limit, which may cause issues with Logstash parsing.

HTTP

```
// Event: HTTP_RESPONSE
if (HTTP.uri != null) {
    RemoteSyslog.info(JSON.stringify({
        'eh_event'     : 'http',
        'src_ip'       : Flow.client.ipaddr.toString(),
        'dest_ip'      : Flow.server.ipaddr.toString(),
        'uri'          : HTTP.uri,
        'req_size'     : HTTP.reqSize,
        'req_L2bytes'  : HTTP.reqL2Bytes,
        'rsp_L2bytes'  : HTTP.rspL2Bytes,
        'status_code'  : HTTP.statusCode,
        'tprocess'     : HTTP.tprocess,
        'req_rtos'     : HTTP.reqRTO,
        'rsp_rtos'     : HTTP.rspRTO
    }));
}
```

DNS

```
// Event: DNS_RESPONSE
var msg = {
    'eh_event': 'dns',
    'src_ip'       : Flow.client.ipaddr.toString(),
    'dest_ip'      : Flow.server.ipaddr.toString(),
    'qname' : DNS.qname,
    'qtype' : DNS.qtype,
    'opcode' : DNS.opcode,
    'tprocess' : DNS.tprocess
};
// If you want every answer, build loop here
var answer = DNS.answers[0];

if (answer !== undefined) {
        msg.ans_name = answer.name;
        msg.ans_ttl = answer.ttl;
        msg.ans_type = answer.type;

    if (answer.data !== null) {
        msg.ans_data = answer.data;
    }
}
```

```
if (DNS.error !== null) {
    msg.error = DNS.error;
}
RemoteSyslog.info(JSON.stringify(msg));
```

Database

```
// Event: DB_REQUEST, DB_RESPONSE
// Set to true for full SQL statements
var SHOW_FULL_STATEMENT = false;

if (event == 'DB_REQUEST' && SHOW_FULL_STATEMENT) {
    Flow.store.db_statement = DB.statement || DB.procedure;
}

else if (event == 'DB_RESPONSE') {
    var msg = {
        'eh_event': 'database',
        'method' : DB.method,
        'user' : DB.user,
        'req_L2bytes' : DB.reqL2Bytes,
        'rsp_L2bytes' : DB.rspL2Bytes
    }
    if (Flow.store.db_statement !== null && SHOW_FULL_STATEMENT) {
        msg.query = Flow.store.db_statement;
        Flow.store.db_statement = null;
    }

    if (DB.error != null) {
        msg.error=DB.error;
    }

    if (DB.tprocess) {
        msg.tprocess = DB.tprocess;
    }

    RemoteSyslog.info(JSON.stringify(msg));
}

else {
    debug('error');
}
```

Storage

```
// Event: CIFS_RESPONSE
var msg = {
    "eh_event": "cifs",
    "method" : CIFS.method,
    "user" : CIFS.user,
    "access_time" : CIFS.accessTime,
    "req_L2bytes" : CIFS.reqL2Bytes,
    "rsp_L2bytes" : CIFS.rspL2Bytes};

if (CIFS.resource != null) {
    msg.filename = CIFS.resource;
}
    if (CIFS.error != null) {
  msg.error= CIFS.error;
}
RemoteSyslog.info(JSON.stringify(msg));
```