Construya su propia herramienta de monitoreo de errores

por Bartłomiej Pasik MVB · 12 y 18 de marzo · Zona de rendimiento · Tutorial

En este tutorial, describiré cómo puedes crear tu propio vigilante de errores. La versión final de este proyecto se puede encontrar en GitHub . Más adelante me referiré al código de este repositorio, para que pueda verificarlo.

¿Por qué utilizar una herramienta de supervisión de errores más simple?

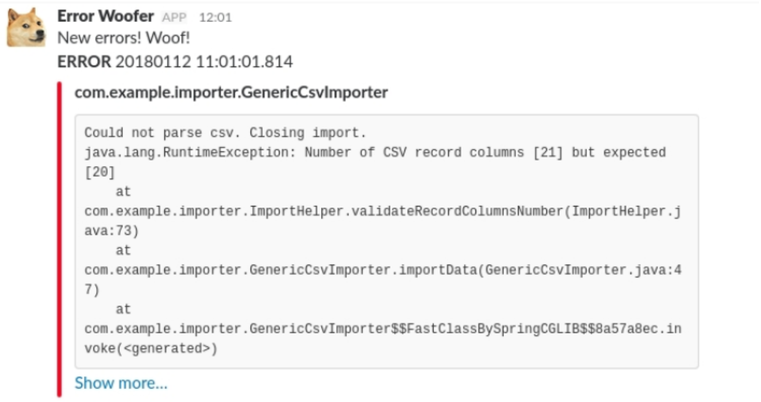
Si su aplicación está en producción o lo estará en un futuro próximo, debe buscar algún tipo de herramienta de supervisión de errores. De lo contrario, tendrás un gran problema. Y como desarrollador, déjeme decirle que buscar errores manualmente en su entorno de producción no es genial.

Encuentre la causa del problema antes de que sus clientes lo noten

Por ejemplo, supongamos que su aplicación está realizando un procesamiento en segundo plano que no es visible a primera vista para el usuario final. El proceso falla en uno de los pasos de fondo. Si tiene una herramienta de monitoreo de errores, tendrá la posibilidad de corregir el error antes de que sus clientes lo noten.

Reduzca el tiempo de búsqueda para corregir el tiempo

Sin una herramienta de monitoreo, cuando se informa un error, su equipo probablemente comenzará a buscar manualmente los registros. Esto extiende significativamente el tiempo de reparación. Ahora imagine que su equipo recibe una notificación de inmediato cuando aparece el error: ahora puede omitir esa parte que consume mucho tiempo.



Error Woofer APP 12:01
New errors! Woof!
ERROR 20180112 11:01:01.814

com.example.importer.GenericCsvImporter

```
Could not parse csv. Closing import.
java.lang.RuntimeException: Number of CSV record columns [21] but expected
[20]
    at
com.example.importer.ImportHelper.validateRecordColumnsNumber(ImportHelper.j
ava:73)
    at
com.example.importer.GenericCsvImporter.importData(GenericCsvImporter.java:4
7)
    at
com.example.importer.GenericCsvImporter$$FastClassBySpringCGLIB$$8a57a8ec.in
voke(<generated>)
```

Show more...

Infraestructura de monitoreo

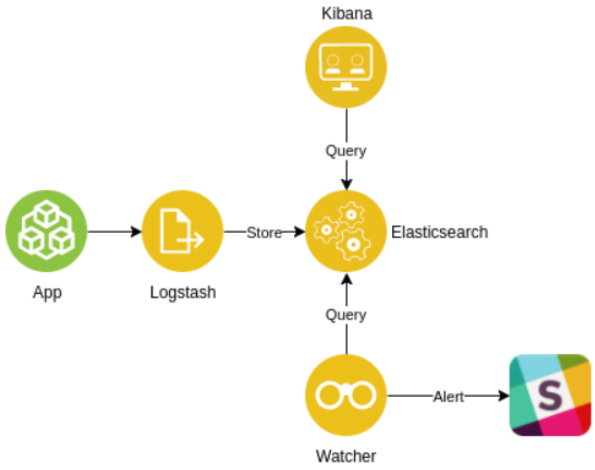
En este tutorial, usaremos la pila Elasticsearch + Logstash + Kibana para monitorear nuestra aplicación. ELK es gratuito cuando utiliza suscripciones de Código Abierto y Básico . Si desea utilizar funcionalidades personalizadas, es decir, alertas, seguridad, aprendizaje automático, tendrá que pagar.

Desafortunadamente, las alertas no son gratis. Si desea enviar un mensaje de alerta a un canal de Slack o enviar un correo electrónico a alguien sobre un error crítico, deberá usar X-Pack "semi-pago". Algunas partes son gratuitas en la suscripción Básica.

Sin embargo, podemos implementar nuestro propio observador para eludir los altos costos de Elastic.

Tengo buenas noticias para ti; ya las he implementado para ti. Volveremos sobre eso más tarde.

La siguiente imagen describe cómo se verá nuestra infraestructura.



Logstash lee los registros, extrae la información que queremos y luego envía los datos transformados a Elasticsearch.

Consultaremos Elasticsearch para los registros recientes que contengan el nivel de registro de errores utilizando nuestro vigilante Node.js Elasticsearch personalizado. The Watcher enviará mensajes de alerta a un canal Slack cuando la consulta arroje algunos resultados. La consulta se ejecutará cada 30 s.

Kibana es opcional aquí; sin embargo, está incluido en el repositorio así que si desea analizar los registros de la aplicación de una manera elegante, aquí tiene. No lo describiré en este artículo, así que visite el sitio de Kibana para ver qué puede hacer con él.

Pila ELK apilada

La configuración manual de Elasticsearch, Logstash y Kibana es bastante aburrida, por lo que utilizaremos una versión ya cargada. Para ser más precisos, usaremos el repositorio Docker ELK que contiene lo que necesitamos. Modificaremos este repositorio para cumplir con nuestros requisitos, por lo tanto, clonarlo y seguir el artículo o navegar en el repositorio final .

Nuestras necesidades:

- Lectura de registros de archivos
- Análisis de registros personalizados de Java
- Análisis de marca de tiempo personalizada

Estamos utilizando Logback en nuestro proyecto y tenemos un formato de registro personalizado. A continuación, puede ver la configuración del apilador de Logback:

```
< appender  name = "stdout"  class = "ch.qos.lc
        < codificador >
            < patrón > % d {aaaaMMdd HH: mm: ss
        </ encoder >
</ appender >
```

Aquí están los registros de muestra:

```
20180107 12: 03: 26.353 [pool-48-thread-1] DEBU
20180122 11: 12: 09.541 [http-nio-8081-exec-73]
java.lang.RuntimeException: Número de columnas
        at com.example.service.importer.ImportH
        at com.example.service.importer.Generic
        at com.example.service.importer.Generic
        at org.springframework.cglib.proxy.Meth
        at org.springframework.aop.framework.Cg
        at com.example.service.runnerarea.impor
        at com.example.ui.common.window.ImportW
        at com.example.ui.common.window.ImportW
        at sun.reflect.GeneratedMethodAccessor1
        at sun.reflect.DelegatingMethodAccessor
        at java.lang.reflect.Method.invoke(Meth
        at com.not.a.vaadin.event.ListenerMetho
        at com.not.a.vaadin.event.EventRouter.f
```

```
	at com.not.a.vaadin.event.EventRouter.f
	at com.not.a.vaadin.server.AbstractClie
	at com.not.a.vaadin.ui.Upload.fireUploa
	at com.not.a.vaadin.ui.Upload$2.streami
	at com.not.a.vaadin.server.communicatic
	at com.not.a.vaadin.server.communicatic
	at com.not.a.vaadin.server.communicatic
	at com.not.a.vaadin.server.communicatic
	at com.not.a.vaadin.server.VaadinServic
	at com.not.a.vaadin.server.VaadinServle
	at javax.servlet.http.HttpServlet.servi
	at org.apache.catalina.core.Applicatior
	at org.apache.catalina.core.Applicatior
	at org.apache.tomcat.websocket.server.W
	at org.apache.catalina.core.Applicatior
	at org.apache.catalina.core.Applicatior
	at org.apache.catalina.core.StandardWra
	at org.apache.catalina.core.StandardCor
	at org.apache.catalina.authenticator.Au
	at org.apache.catalina.core.StandardHos
	at org.apache.catalina.valves.ErrorRepo
	at org.apache.catalina.valves.AbstractA
	at org.apache.catalina.core.StandardEng
	at org.apache.catalina.connector.Coyote
	at org.apache.coyote.http11.Http11Proce
	at org.apache.coyote.AbstractProcessorL
```

```
        at org.apache.coyote.AbstractProtocol$C
        at org.apache.tomcat.util.net.NioEndpoi
        at org.apache.tomcat.util.net.SocketPrc
        at java.util.concurrent.ThreadPoolExecu
        at java.util.concurrent.ThreadPoolExecu
        at org.apache.tomcat.util.threads.TaskT
        at java.lang.Thread.run(Thread.java:748
```

Firstly, we need to update the docker-compose.yml file to consume our logs directory and custom patterns for Logstash. The Logstash service needs two extra lines in its volume section:

```
- ./logstash/patterns:/usr/share/logstash/patte
- $LOGS_DIR:/usr/share/logstash/logs
```

The first line binds our pattern directory. The second attaches logs to container. *$LOGS_DIR* variable will later be added to an *.env* file, which will give us the ability to change logs dir without modifying the repository. That's all we need.

If you'd like to persist data between container restarts, you can bind Elasticsearch and Logstash directories to some directory outside the Docker.

Here's the *.env* file. You can replace *logs dir* with your path.

```
ELK_VERSION=5.6.3
NODE_VERSION=9.3.0
LOGS_DIR=./logs
```

How to Configure Logstash to Consume App Logs

Logstash's pipeline configuration can be divided into three sections:

- **Input**: Describes sources which Logstash will be consuming.

- **Filter**: Processes logs, i.e. data extraction, transformation.
- **Output**: Sends data to external services

```
input {
        file {
                path => "/usr/share/logstash/lo
                start_position => "beginning"
                codec => multiline {
                        patterns_dir => ["./pat
                        pattern => "^%{MY_TIMES
                        negate => true
                        what => "previous"
                }
        }
}

filter {
        grok {
                patterns_dir => ["./patterns"]
                match => { "message" => "%{MY_T
                overwrite => [ "message" ]
        }
        date {
                match => [ "customTimestamp" ,
                remove_field => [ "timestamp" ]
        }
}

output {
        elasticsearch {
                hosts => "elasticsearch:9200"
        }
}
```

The code above is the full Logstash configuration.

The input section is quite simple. We define basic input properties such as logs path and logs beginning position when starting up Logstash. The most position when starting up Logstash. The most interesting part is the codec where we configure handling multiline Java exceptions. It will look up for beginning by, in our example, a custom timestamp and it will treat all text after till next custom timestamp as one log entry (document in Elasticsearch).

I've included a patterns directory, so we can use our custom pattern in multiline regex. It's not required, you can use normal regex here.

The filter section is the most important part of a Logstash configuration. This is the place where the magic happens. Elastic defined plenty of useful plugins which we can use to transform log events. One of them is Grok, which we'll use in the monitoring tool.

Grok parses and structures text, so you can grab all fields from your logs, i.e. timestamp, log level, etc. It works like regex. Just define your log pattern with corresponding field names and Grok will find matching values in the event. You can use default Grok patterns or create your own to match custom values.

In our example, we'll use a custom timestamp, so we need to define a custom pattern. Grok allows us to use custom patterns ad hoc in message pattern. However, we want to use it more than once, so we defined a patterns file which we can include in places where we need the pattern e.g. multiline codec and Grok. If you use a standard timestamp format, just use the default one.

Here's the pattern file:

```
MY_TIMESTAMP %{YEAR}%{MONTHNUM}%{MONTHDAY} %{TI
```

The file structure is the same as in other Grok patterns. The first word in the line is the pattern name and rest is the pattern itself. You can use default patterns while defining your pattern. You can also use Regex, if none of the default matches your needs. In our case, the log format is e.g. 20180103 00:01:00.518, so we're able to use already defined patterns.

In **the output section**, we define that transformed logs will be sent to the Elasticsearch.

Docker File Permissions

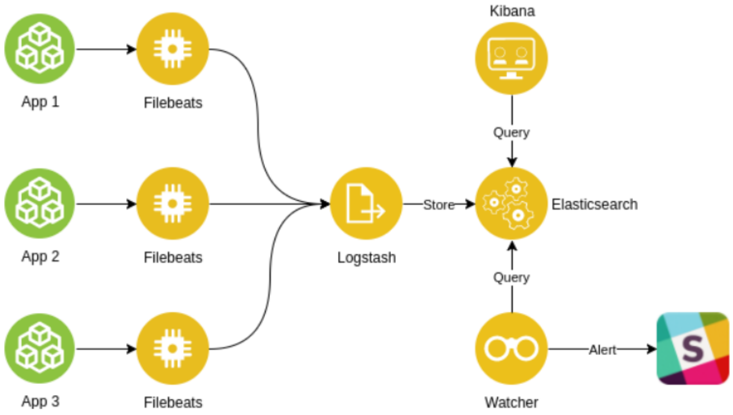
One thing that took me some time to figure out was the configuration of the file permissions of the logs accessed by dockerized Logstash.

If your logs are created as a user with ID 1000, you won’t notice the problem and you can skip this step. However, you’re most likely dealing with quite the opposite. For example, you run your application on Tomcat and the logs are created by the Tomcat user and then bound as a volume to the Logstash container. The Tomcat user is not the first user (1000) in the system, so user id won’t match in the container. Default Logstash image runs as a user 1000, so it can read logs only with permission of user 1000. It doesn’t have access to other users’ files.

```
ARG ELK_VERSION

# https://github.com/elastic/logstash-docker
FROM docker.elastic.co/logstash/logstash:${ELK_

# Add your logstash plugins setup here
# Example: RUN logstash-plugin install logstash

USER root

RUN groupadd --gid 1001 tomcat

RUN usermod -a -G tomcat logstash

USER logstash
```

The trick here is to switch to root in Docker file and create a new group with an ID matching the log creator group id and then add the Logstash user to it. Then we add the user which runs the container to the group which owns the logs on the server (*sudo usermod -a -G <group> <user>*). After that, we switch back to the Logstash user for security reasons to secure the container.

Filebeats: Log Agent



The implementation described so far can be used for one application. We could scale it to support many applications, but it wouldn't be fun nor easy. Logstash reads lines one by one and sends them after transformation to Elasticsearch.

I've got a better solution for you. Elastic created family software called the Beats. One of them is Filebeats, which kills the pain of log access. The Filebeats is simply a log shipper. It takes logs from the source and transfers them to Logstash or directly to Elasticsearch. With it, you can forward your data, although it can also do some of the things what Logstash does, e.g. transforming logs, dropping unnecessary lines, etc. But, Logstash can do more.

If you have more than one application or more than one instance of the application, then Filebeats is for you. Filebeats transfers logs directly to the Logstash to the port defined in the configuration. You just define where should they look for logs and you define the listening part in the Logstash.

The file permission problem will, of course, be present if you want to run the dockerized version of the Filebeats, but that's the cost of virtualization.

I suggest you use Filebeats for production purposes. You will be able to deploy ELK on the server which won't be actually the prod server. Without Filebeats (with Logstash only) you'll need to place it on the same machine where the logs reside.

Sending Slack Alert

Sending Slack Alert Message

Elastic delivers the Watcher functionality within X-Pack, bundled into the Elasticsearch and, what's more, there is an already defined Slack action which can send custom messages to your Slack channel, not to mention more actions. However, as stated before, it's not free. The Watcher is available in Gold subscription, so if that's ok for you, then you can skip rest of the article. If not, let's go further.

When I noticed that the Elastic Watcher is a paid option, I thought that I could do my own watcher which would send alert messages to Slack. It's just a scheduled job which checks if there's something to send, so it shouldn't be hard, right?

The Watcher

I created an npm package called Elasticsearch Node.js Watcher, which does the basics of what the X-Pack's Watcher does, namely watching and executing actions when specified conditions are satisfied. I chose Node.js for the the Watcher, because it's the easiest and fastest option for a small app which would do all of the things I need.

This library takes two arguments when creating a new instance of a watcher:

- **Connection configuration**: It defines connection parameters to Elasticsearch. Read more about it in the documentation.
- **The Watcher configuration**: Describes when and what to do if there's a hit from Elasticsearch. It contains five fields (one is optional):
    - **Schedule**: The Watcher uses it to schedule cron job.
    - **Query**: Query to be executed in Elasticsearch, the result of which will be forward to predicate and action.
    - **Predicate**: Tells if action should be executed.
    - **Action**: Task which is executed after

satisfying predicate.

- **Error handler (optional)**: Task which will be executed when an error appears.

We need to create a server which would start our Watcher, so let's create *index.js* with Express server. To make the environment variables defined in *.env* file visible across Watcher's files, let's also include *dotenv* module.

```
require('dotenv').config();
const express = require('express');
const watch = require('./watcher');

const app = express();

app.listen((err) => {
    if (err) {
        return console.log('something bad happe
    }
    watch();
});
```

The meat. In our configuration, we defined to query Elasticsearch every 30 seconds using the cron notation. In the query field, we defined the index to be searched. By default, Logstash creates indexes named *logstash-<date>*, so we set it to *logstash-** to query all existing indices.

```
const elasticWatcher = require("elasticsearch-r
const sendMessage = require("./slack");

const connection = {
    host: process.env.ELASTICSEARCH_URL,
    log: process.env.ELASTICSEARCH_LOG_LEVEL
};

const watcher = {
    schedule: "*/30 * * * * *",
    query: {
        index: 'logstash-*',
        body: {
            query: {
                bool: {
```

```
        bool: {
          must: {match: {loglevel: "E
          filter: {
            range: {"@timestamp": {
          }
        }
      }
    }
  },
  predicate: ({hits: {total}}) => total > 0,
  action: sendMessage
};

module.exports = () => elasticWatcher.schedule(
```

To find logs, we use Query DSL in the query field. In the example, we're looking for entries with Error log level which appeared in last 30 seconds. In the predicate field, we'll define the condition of hits number at greater than 0 since we don't want to spam Slack with empty messages. The action field references the Slack action described in the next paragraph.

Slack Alert Action

To send a message to a Slack channel or a user, we needed to set up an incoming webhook integration first. As a result, you'll get a URL that you should put it in the Watcher's *.env* file:

```
SLACK_INCOMING_WEBHOOK_URL=<place_here_your_url
ELASTICSEARCH_URL=http://elasticsearch:9200
ELASTICSEARCH_LOG_LEVEL=trace
```

Ok, the last part. Here, we're sending a POST request to Slack's API containing a JSON with a formatted log alert. There's no magic here. We're just mapping Elasticsearch hits to message attachments and adding some colors to make it fancier. In the title, you can find information about the class of the error and the timestamp. See how you can format your messages.

```
const request = require('request');

const RED = '#ff0000';

const sendMessage = (message, channels) => {
    console.log('Sending message to Slack');

    const cb = (err, response, body) => {
        if (err) {
            console.log('Error appeared while s
        }
        console.log('Message sent', body);
    };

    const sendRequest = (message) =>
        request({url: process.env.SLACK_INCOMIN

    if (channels) {
        channels.forEach(channel => {
            message.channel = channel;
            sendRequest(message);
        });
    } else {
        sendRequest(message);
    }
};

const send = (data) => {
    const mapHitToAttachment = (source) => (
        {
            pretext: `*${source.loglevel}* ${sc
            title: `${source.class}`,
            text: `\`\`\`${source.msg}\`\`\``,
            color: RED,
            mrkdwn_in: ['text', 'pretext']
        }
    );

    const message = {
        text: "New errors! Woof!",
        attachments: data.hits.map(hit => mapHi
    };


    sendMessage(message);
};

module.exports = send;
```

Dockerization

Finally, we'll dockerize our Watcher, so here's the code of Dockerfile:

```
ARG NODE_VERSION

FROM node:${NODE_VERSION}

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

CMD [ "npm", "start" ]
```

For development purposes of the Watcher, that's enough. ELK will keep running and you'll be able to restart the Watcher server after each change. For production, it would be better to run the Watcher alongside ELK. To run the prod version of the whole infrastructure, let's add a Watcher service to docker-compose file. Service needs to be added to the copied docker-compose file with a *-prod* suffix.

```
  watcher:
      build:
        context: watcher/
        args:
          NODE_VERSION: $NODE_VERSION
      networks:
        - elk
      depends_on:
        - elasticsearch
```

Then, we can start up our beautiful log monitor with one docker-compose command.

one docker-compose command.

```
docker-compose -f docker-compose-prod.yml up -c
```

In the final repository version, you can just execute make run-all command to start the prod version. Check out the Readme.md; it describes all needed steps.

But…

This solution is the simplest one. I think the next step would be to aggregate errors. In the current version, you'll get errors one by one in your Slack channel. This is good for dev/stage environment because they're used by few users. Once you're on production, you'll need to tweak Elasticsearch's query; otherwise, you'll be flooded with messages. I'll leave it to you as homework

You need to analyze the pros and cons of setting up all of this by yourself. On the market, we have good tools such as Rollbar or Sentry, so you need to choose if you want to use the "Free" (well, almost, because some work needs to be done) or the Paid option.

I hope you found this article helpful.

Topics: PERFORMANCE , MONITORING , TUTORIAL

Published at DZone with permission of Bartłomiej Pasik , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.