Construya su propia herramienta de monitoreo de errores

por Bartłomiej Pasik MVB · Mar. 12, 18 · Performance Zone · Tutorial

Maintain Application Performance with real-time monitoring and instrumentation for any application. Learn More!

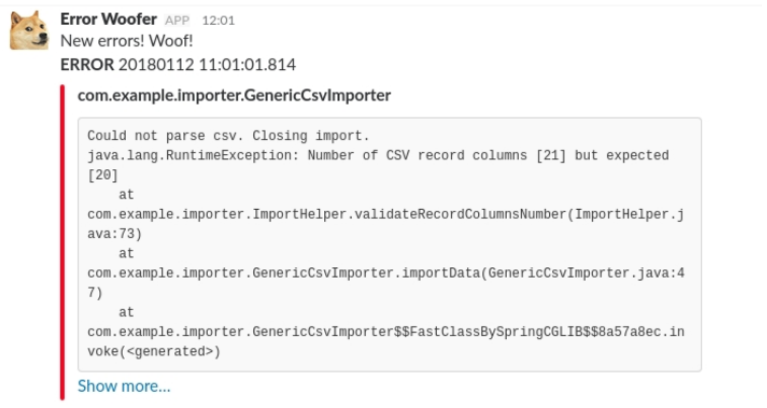
In this tutorial, I'll describe how you can create your own error watcher. The final version of this project can be found on GitHub. I'll later refer to the code from this repository, so you can check it out.

Why Use a Simpler Error Monitoring Tool?

If your application is on production or will be in near future, you should look for some kind of error monitoring tool. Otherwise, you'll have a really big problem. And as a developer, let me tell you that looking for errors manually in your production environment isn't cool.

Find the Cause of the Problem

away when the error appears — you can now skip that time-consuming part.


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...

Monitoring infrastructure

In this tutorial, we'll use the Elasticsearch + Logstash + Kibana stack to monitor our application. ELK is free when you use Open Source and Basic subscriptions. If you want to use custom functionalities, i.e. alerting, security, machine learning, you'll need to pay.

Unfortunately, alerting isn't free. If you'd like to send an alert message to a Slack channel or email someone about a critical error, you'll need to use "semi-paid" X-Pack. Some parts are free in the Basic subscription.

However, we can implement our own watcher to bypass Elastic's high costs. I've got a good news for you — I've implemented it for you already. We'll get back to it later.

The image below describes how our infrastructure is going to look like.

Elasticsearch.

We will query Elasticsearch for recent logs containing error log level using our custom Node.js Elasticsearch Watcher. The Watcher will send alert messages into a Slack channel when the query returns some results. The query will be executed every 30s.

Kibana is optional here; however, it's bundled in the repo so if you'd like to analyze application logs in some fancy way, here you go. I won't describe it in this article, so visit the Kibana site to see what you can do with it.

Dockerized ELK Stack

Setting up Elasticsearch, Logstash, and Kibana manually is quite boring, so we'll use an already Dockerized version. To be more precise, we'll use Docker ELK repository which contains what we need. We'll tweak this repo to meet our requirements, so either clone it and follow the article or browse the final repo.

Our requirements:

- Reading logs from files
- Parsing custom Java logs
- Parsing custom timestamp

We're using Logback in our project and we have a custom log format. Below, you can see the Logback appender configuration:

```
        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$2.streami
        at com.not.a.vaadin.server.communicatio
        at org.apache.tomcat.websocket.server.W
        at org.apache.catalina.core.Applicatior
        at org.apache.catalina.core.Applicatior
        at org.apache.catalina.core.StandardWra
        at org.apache.catalina.core.StandardCor
        at org.apache.catalina.authenticator.Au
        at org.apache.catalina.core.StandardHos
        at org.apache.catalina.valves.ErrorRepc
        at org.apache.catalina.valves.AbstractA
        at org.apache.catalina.core.StandardEng
        at org.apache.catalina.connector.Coyote
        at org.apache.coyote.http11.Http11Proce
        at org.apache.coyote.AbstractProcessorL
        at org.apache.coyote.AbstractProtocol$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
```

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/lc
                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 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

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.

```
# 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

Logstash de usuario
```

El truco aquí es cambiar a raíz en el archivo Docker y crear un nuevo grupo con un ID que coincida con el ID del grupo creador de registros y luego agregarle el usuario Logstash. Luego agregamos el usuario que ejecuta el contenedor al grupo que posee los registros en el servidor (*sudo usermod -a -G <group> <user>*). Después de eso, volvemos al usuario de Logstash por razones de seguridad para asegurar el contenedor.

Filebeats: Log Agent

Kibana
Query

Descargar My Free PDF

registro. Filebeats es simplemente un cargador de registro. Toma registros de la fuente y los transfiere a Logstash o directamente a Elasticsearch. Con él, puede reenviar sus datos, aunque también puede hacer algunas cosas que Logstash hace, por ejemplo, transformar registros, eliminar líneas innecesarias, etc. Pero, Logstash puede hacer más.

Si tiene más de una aplicación o más de una instancia de la aplicación, Filebeats es para usted. Filebeats transfiere registros directamente al Logstash al puerto definido en la configuración . Simplemente defina dónde deberían buscar los registros y defina la parte de escucha en el Logstash.

El problema de permiso de archivo, por supuesto, estará presente si desea ejecutar la versión dockada de Filebeats, pero ese es el costo de la virtualización.

Sugiero que use Filebeats para fines de producción. Podrá implementar ELK en el servidor, que en realidad no será el servidor prod. Sin Filebeats (solo con Logstash), deberá colocarlo en la misma máquina donde residen los registros.

Envío de mensaje de alerta de holgura

Elastic ofrece la funcionalidad Watcher dentro de X-Pack, incluida en Elasticsearch y, además, hay una acción Slack ya definida que puede enviar mensajes personalizados a su canal Slack, sin mencionar más hace el Observador del X-Pack, es decir, ver y ejecutar acciones cuando se cumplen las condiciones especificadas. Elegí Node.js para The Watcher, porque es la opción más fácil y rápida para una aplicación pequeña que haría todas las cosas que necesito.

Esta biblioteca toma dos argumentos al crear una nueva instancia de un observador:

- **Configuración de conexión** : define los parámetros de conexión a Elasticsearch. Lea más sobre esto en la documentación .
- **La configuración de Watcher** : describe cuándo y qué hacer si hay un golpe de Elasticsearch. Contiene cinco campos (uno es opcional):
    - **Programa** : The Watcher lo usa para programar el trabajo cron.
    - **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

```
    }
    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: {
                    must: {match: {loglevel: "E
                    filter: {
                        range: {"@timestamp": {
```

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');
```

```
        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);
};
```

```

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.

```
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.

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

Recursos para socios de **rendimiento**

Microservices Zone · Tutorial

Learn the Benefits and Principles of Microservices Architecture for the Enterprise

In part 5, we'll get our "accountservice" up and running on a locally deployed Docker Swarm cluster and discuss the core concepts of container orchestration.

This blog post deals with the following:

- Docker Swarm and container orchestration
- Containerize our accountservice using Docker
- Setting up a local Docker Swarm cluster
- Deploying the accountservice as a Swarm Service
- Run the benchmarks and collect metrics

Gophers beware: After writing this part of the blog series, I realized there's nothing Go-specific in this part. I hope you'll enjoy it anyway.

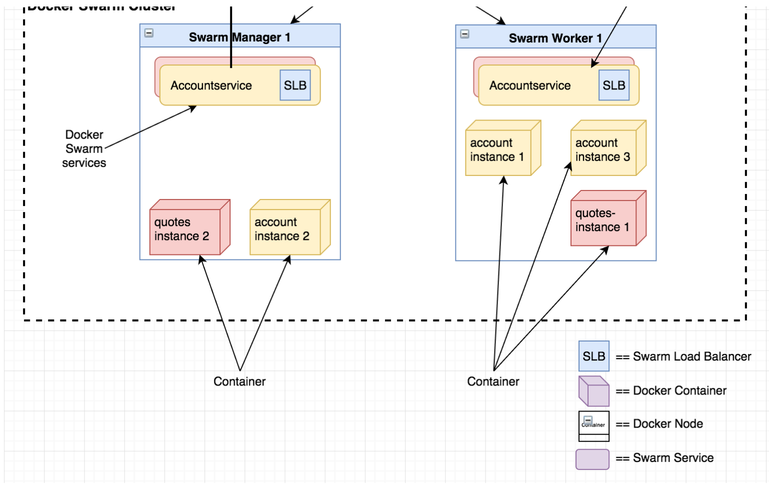
Before going practical, a quick introduction of the concept of a "container orchestrator" may be of use.

As an application becomes more complex and has to handle an (hopefully) ever higher load, we have to deal with the fact that we could be running hundreds of service instances spread out over a lot of physical (or virtualized) hardware. Container orchestration different nomenclature and hierarchy of key abstractions, but the concept, on the whole, is roughly the same.

The container orchestrator not only handles the lifecycle of our service for us, it also provides mechanics for things like service discovery, load-balancing, internal addressing, and logging.

In Docker Swarm, there are three concepts that entail an introduction:

- Node: A node is an instance of the Docker engine participating in the swarm. Technically, see it as a host having its own CPU resources, memory and network interface. A node can be either a *manager node* or a *worker node*.
- Service: A service is the definition of what to execute on the worker nodes as defined by a container image and the commands you instruct the container(s) to execute. A service can be either *replicated* or *global*, see the docs for details. A service can be seen as an abstraction for letting an arbitrary number of containers form a logical "service" accessible through its name throughout and possibly outside the cluster without having to know anything about the internal network topology of the environment.
- Task (e.g. container): For all practical means, consider a task a Docker container. The Docker



Source Code

While this part doesn't change any of the Go code from the previous parts, we're going to add some new files for running on Docker. Feel free to check out the appropriate branch from git to get the source of the completed state of this part.

```
git checkout P5
```

Docker Installation

For this part to work, you'll need to have Docker installed on your dev machine. I suggest following this guide for your Operating System. Personally, I've used Docker Toolbox with Virtualbox when developing the code for this blog series, but you could just as well use Docker for Mac/Windows or run Docker natively on Linux.

Creating a Dockerfile

building our own image from. iron/base is a very compact image well suited for running a Go application.

- EXPOSE - defines a port number we want to expose on the internal Docker network so it becomes reachable.
- ADD - Adds a file *accountservice-linux-amd64* to the root (/) of the container filesystem.
- ENTRYPOINT - defines what executable to run when Docker starts a container of this image.

Building for Another CPU Architecture/Operating System

As you see, the file we're adding has this "linux-amd64" in its name. While we can name a Go executable just about anything, I like using a convention where we put the OS and target CPU platform into the executable name. I'm writing this blog series on a Mac running OS X. So if I just build a go executable of our accountservice from the command line when in the */goblog/accountservice* folder:

```
> go build
```

That will create an executable called *accountservice* in the same folder. The problem with that executable is that it won't run in our Docker container as the

Since both OS X and our Linux-based container runs on the AMD64 CPU architecture we don't need to set (and reset) the GOARCH env var. But if you're building something for a 32-bit OS or perhaps an ARM processor you would need to set GOARCH appropriately before building.

Creating a Docker Image

Now it's time to build our very first Docker image containing our executable. Go to the parent folder of the */accountservice* folder, it should be *$GOPATH/src/github.com/callistaenterprise/goblog*.

When building a Docker container image, we usually tag it with a name usually using a [prefix]/[name] naming convention. I typically use my github username as prefix, e.g. *eriklupander/myservicename*. For this blog series, I'll use "someprefix". Execute the following command from the root project folder (e.g.*/goblog*) to build a Docker image based on the Dockerfile above:

```
> docker build -t someprefix/accountservice acc

Sending build context to Docker daemon 13.17 ME
Step 1/4 : FROM iron/base
 ---> b65946736b2c
Step 2/4 : EXPOSE 6767
 ---> Using cache
```

be running with multiple nodes or if we'd want to share our new image, we'd use docker push to make the image available for hosts other than our currently active Docker Engine provider host to pull.

We can now try to run this image directly from the command line:

```
> docker run --rm someprefix/accountservice
Starting accountservice
Seeded 100 fake accounts...
2017/02/05 11:52:01 Starting HTTP service at 67
```

However - note that this container is *not* running on your host OS localhost anymore. It now lives in its own networking context and we can't actually call it directly from our host operating system. There are ways to fix that of course, but instead of going down that route we'll now set up Docker Swarm locally and deploy our "accountservice" there instead.

Use Ctrl+C to stop the container we just started.

One of the goals of this blog series is that we want to run our microservices inside a container orchestrator. For many of us, that typically means Kubernetes or Docker Swarm. There are other orchestrators as well such as Apache Mesos and Apcera, but this blog series will focus exclusively on Docker Swarm based on Docker 1.13.

The tasks involved when setting up a Docker Swarm important thing is that after this section you need to have a working Swarm Manager up and running.

The example here uses docker-machine and makes a virtual Linux machine running on Virtualbox my Swarm Manager. It's referred to as "swarm-manager-1" in my examples. You can also take a peek at the official documentation on how to create a Swarm.

This command initializes the docker-machine host identified as *swarm-manager-1* as a swarm node with the IP address of the same *swarm-manager-1* node:

```
> docker $(docker-machine config swarm-manager-
```

If we'd be creating a multi-node swarm cluster we'd make sure to store the *join-token* emitted by the command above as we'd need it later if we were to add additional nodes to the swarm.

Create an Overlay Network

A docker overlay network is a mechanism we use when adding services such as our "accountservice" to the Swarm so it can access other containers running in the same Swarm cluster without having to know anything about the actual cluster topology. Let's create such a network:

```
docker network create --driver overlay my_netwc
```

my_network is the name we gave the network.

This is also the name other services will use when addressing our service within the cluster. So if you had another service that would like to call the *accountservice*, that service would just do a GET to *http://accountservice:6767/accounts/10000*

- -replicas: The number of instances of our service we want. If we'd have a multi-node Docker Swarm cluster the swarm engine would automatically distribute the instances across the nodes.
- -network: Here we tell our service to attach itself to the overlay network we just created.
- -p: Maps [internal port]:[external port]. Here we used 6767:6767 but if we'd created it using 6767:80 then we would access the service from port 80 when calling externally. Note that this is the mechanism that makes our service reachable from outside the cluster. Normally, you shouldn't expose your services directly to the outside world. Instead, you'd be using an EDGE-server (e.g. a reverse proxy) that would have routing rules and security setup so external consumers wouldn't be able to reach your services except in the way you've intended them to.
- someprefix/accountservice: This how we specify *which* image we want the container to run. In our case this is the tag we specified when we created the container. Note! If we'd be running a

Sweet. We should now be able to curl or even use a web browser to query our API. The only thing we need to know up front is the public IP of the Swarm. Even if we're running only one instance of our service on a swarm with many nodes, the overlay network and Docker Swarm lets us ask *any* of the swarm hosts for our service based on its port. That also means that two services cannot expose *the same* port externally. They may very well have the same port *internally* but to the outside world - the Swarm is one.

Anyway - remember that environment variable *ManagerIP* we saved earlier?

```
> echo $ManagerIP
192.168.99.100
```

If you've changed your terminal session since then, you can re-export it:

```
> export ManagerIP=`docker-machine ip swarm-mar
```

Let's curl:

```
> curl $ManagerIP:6767/accounts/10000
{"id":"10000","name":"Person_0"}
```

It's alive!

Deploy a Visualizer

While using the Docker command-line API to examine the state of your Swarm (such as *docker service ls*) is fully viable, a more graphical

```
    --mount=type=bind,src=/var/run/docker.sock,ds
manomarks/visualizer
```

This creates a service we can access at port 8000. Direct your browser to *http://$ManagerIP:8000*:

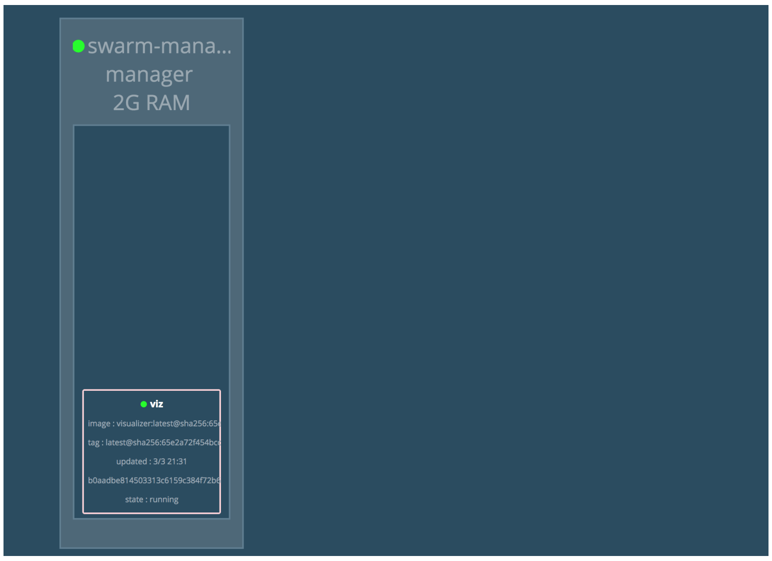


image : visualizer:latest@sha256:65e
tag : latest@sha256:65e2a72f454bcc
updated : 3/3 21:31
b0aadbe814503313c6159c384f72b6
state : running

Bonus Content!

I've made a little Swarm visualizer myself called "dvizz" using Go, the Docker Remote API and D3.js force graphs. You can install it directly from a pre-baked image if you like:

```
docker service create \
  --constraint node.role==manager \
  --replicas 1 --name dvizz -p 6969:6969 \
  --mount type=bind,source=/var/run/docker.soc
  --network my_network \
```

do in part 7 of the blog series. Feel free to branch dvizz or submit a pull request if you want to help to make it even cooler, more useful or less buggy!

Adding the quotes-service

Not much of a Microservice landscape with only one type of service (our ubiquitous accountservice) in it. Let's remedy that by deploying the Spring Boot-based " *quotes-service*" previously mentioned directly from a container image I've pushed to Docker Hub tagged as *eriklupander/quotes-service*:

```
> docker service create --name=quotes-service -
```

If you do a *docker ps* to list running Docker containers we should see it started (or starting):

```
> docker ps
CONTAINER ID    IMAGE                          COM
98867f3514a1    eriklupander/quotes-service "ja
```

Note that we're not exporting a port mapping for this service which means it won't be reachable from outside of the Swarm cluster, only internally on port 8080. We'll integrate with this service later on in Part 7 when we'll be looking at Service Discovery and load-balancing.

If you added "dvizz" to your Swarm, it should show something like this now that we've added the "quotes-service" and "accountservice."

```
#!/bin/bash
export GOOS=linux
export CGO_ENABLED=0

cd accountservice;go get;go build -o accountser

export GOOS=darwin

docker build -t someprefix/accountservice accou

docker service rm accountservice
docker service create --name=accountservice --r
```

This script sets up environment variables so we safely can build a statically linked binary for Linux/AMD64, does this and then runs a few Docker commands to (re)build the image and (re)deploy it as a Docker Swarm service. Saves time when prototyping!

Build scripts for Go is a topic I won't dive into in this blog series. Personally, I like the simplicity of shell scripts, though I've occasionally used a gradle plugin myself and I know good ol' make is quite popular as well.

Footprint and Performance

From now on, all benchmarking and collection of CPU/memory metrics will happen on the service

Memory Usage After Startup

```
> docker stats $(docker ps | awk '{if(NR>1) pri

CONTAINER                                    CF
accountservice.1.k8vyt3dulvng9l6y4mj14ncw9   0.
quotes-service.1.h07fde0ejxru4pqwwgms9qt00   0.
```

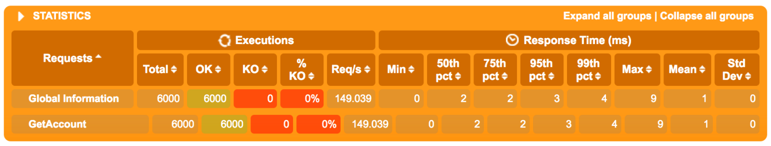
Right after startup, the container consisting of a bare Linux distro and our running "accountservice" uses ~5.6 mb of RAM out of the 2GB I've allocated to the Docker Swarm node. The Java-based quotes-service hovers just under 300 mb - though it should be said that it certainly can be reduced somewhat by tuning its JVM.

CPU and Memory Usage During Load Tests

```
CONTAINER                                    CF
accountservice.1.k8vyt3dulvng9l6y4mj14ncw9   25
```

At 1K req/s within the Swarm running on a there's an overhead involved when the Gatling test that's running on the native OS accesses the Swarm running on a virtualized instance with a bridged network and some perhaps some routing going on within the Swarm and its own overlay network. It's still really nice to be able to serve a peak of 1K req/s at a 4ms mean latency including reading from the BoltDB, serializing to JSON and serving it over HTTP.

In case we need to greatly decrease the number of req/s in later blog posts due to large overhead when adding things like tracing, logging, circuit breakers or whatnot - here's a figure when running 200 req/s:



Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	6000	6000	0	0%	149.039	0	2	2	3	4	9	1	0
GetAccount	6000	6000	0	0%	149.039	0	2	2	3	4	9	1	0

Summary

This sums up part 5 of the blog series where we have learned how to bootstrap a Docker Swarm landscape (with one node) locally and how to package and deploy our "accountservice" microservice as a Docker Swarm Service.

In part 6, we'll add a Healthcheck to our microservice.

Topics: MICROSERVICES, GO, DOCKER SWARM, DEPLOYMENT