Monitoring

Raúl Estrada

Octubre 2020

Monitoring

We have our metrics flowing into Prometheus. We also have a way of exploring and visualizing them. The next step should probably be to configure some sort of alerts, so that we show other people we are doing real work.

Alerting with Prometheus

ALERTING OVERVIEW

Alerting with Prometheus is separated into two parts. Alerting rules in Prometheus servers send alerts to an Alertmanager. The Alertmanager then manages those alerts, including silencing, inhibition, aggregation and sending out notifications via methods such as e-mail, PagerDuty and HipChat.

The main steps to setting up alerting and notifications are:

- Setup and configure the Alertmanager

- Configure Prometheus to talk to the Alertmanager with the-alertmanager.url flag

- Create alerting rules in Prometheus

--https://prometheus.io/docs/alerting/overview/

Let us break this down.

We already have Alertmanager running with some minimal configuration in `/opt/prometheus/alertmanager/alertmanager.yml` .

Our Prometheus instance is aware of it as we passed the `-alertmanager.url=http://localhost:9093` flag.

What is left is to create alerting rules. We'll store these in a `rules/` folder:

Copy

```
# mkdir /opt/prometheus/server/rules
```

We need to tell Prometheus about this location, so we add a `rule_files` section to `prometheus.yml` :

Copy

```
rule_files:
  - "rules/*.rules"
```

This way we can store separate rule files, perhaps based on the type of rules they contain?

As an example, let us have a keepalive and a disk usage alert:

`/opt/prometheus/server/rules/keepalive.rules` :

Copy

```
ALERT Keepalive
  IF up == 0
  FOR 1m
  ANNOTATIONS {
    summary = "Instance {{$labels.instance}} down",
    description = "{{$labels.instance}} of job {{$labels.job}} has been down for more than 1 minute."
  }
```

`/opt/prometheus/server/rules/disk.rules` :

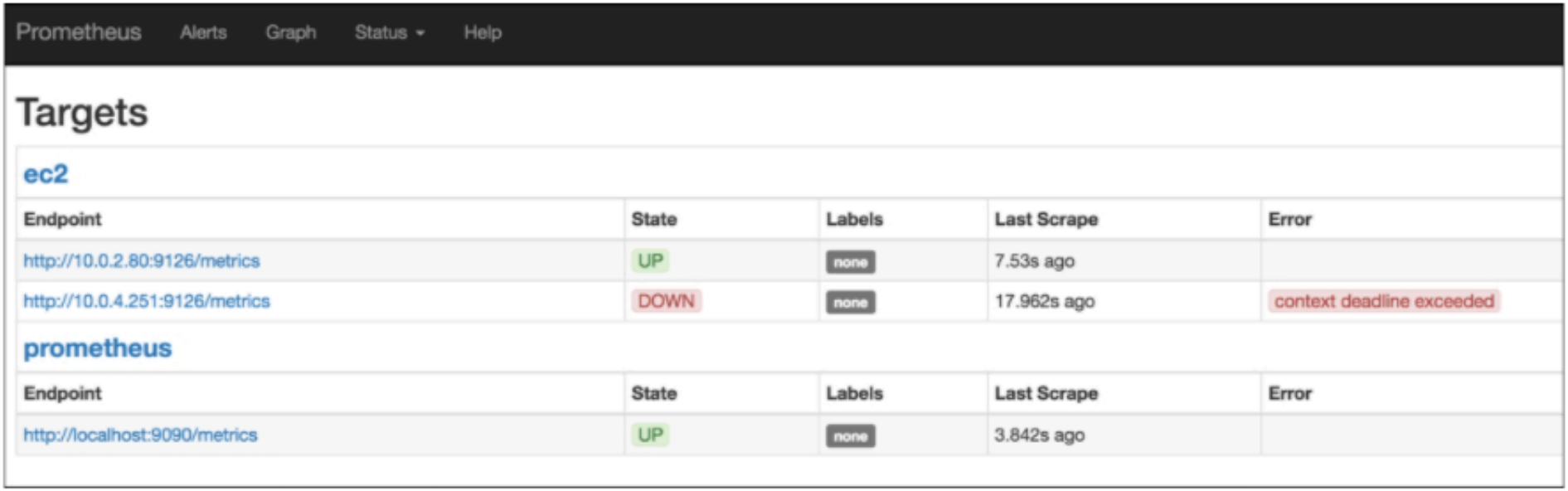
Copy

```
ALERT High_disk_space_usage
  IF disk_used_percent > 20
  FOR 1m
  ANNOTATIONS {
    summary = "High disk space usage on {{ $labels.instance }}",
    description = "{{ $labels.instance }} has a disk_used value of {{ $value }}% on {{ $labels.path }})",
  }
```

As you'll notice, we are being impatient with the `FOR 1m` and `>20`, meaning notifications will fire after just 60 seconds of alert detection and the alert threshold is only 20% of space used.

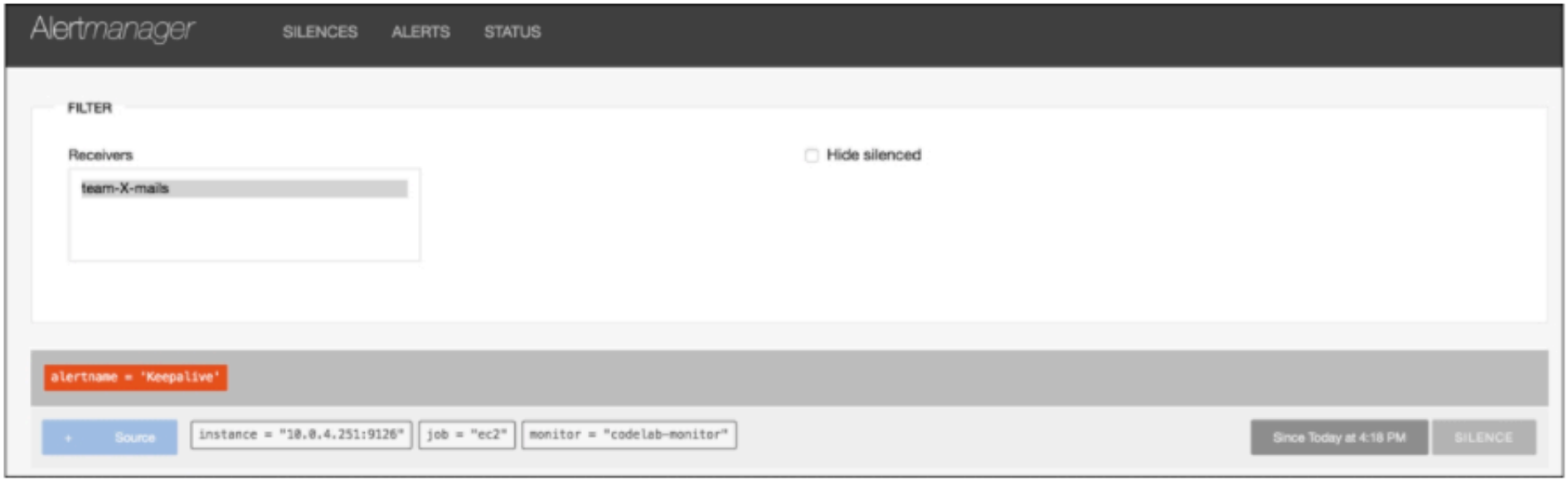
In a more realistic scenario, we should wait a bit longer to filter any transient issues and use severities to distinguish between critical alerts and warnings (ref: https://github.com/prometheus/alertmanager).

Reload Prometheus with the new rules in place. Now let us suppose that one of the web server nodes goes down:



Switching to the **Alerts** tab we see:

In the Alertmanager respectively: (`http://$ public_IP_of_promjenkins_node:9093/#/alerts`):



At this point an e-mail notification should have gone out as well.

Self-remediation with Prometheus and Jenkins

The dream of every operator is an ecosystem that looks after itself.

Imagine for a moment an environment in which, instead of receiving alerts prompting for action, we received mere notifications or reports of actions taken on our behalf.

For example, no more "CRITICAL: Service X is not responding. Please check." but "INFO: Service X was unresponsive at nn:nn:nn and was restarted after N seconds at nn:nn:nn" instead.

Well, technically, this should not be too difficult to achieve if we were to provide enough context to the tools we use today. It is not uncommon to find alerts which tend to get resolved in the same manner under the same conditions and those are to be considered prime candidates for automation.

To demonstrate, let us assume we inherited this old, no longer supported application. A cool app overall, but it does not have the habit of tidying up after itself, so would occasionally fill up its `tmp` directory.

Let us also assume that while we are not particularly excited about having to connect to this app's server to delete `tmp` files at random times of the day, our friend, Mr. Jenkins - does not mind at all.

Conveniently, Jenkins allows jobs to be triggered via a relevant `JOB_URL` and at the same time Prometheus supports webhook calls as a method of alert notification.

Here is the plan:

1. Prometheus will make a webhook call to Jenkins whenever a `disk_space` alert is fired with the alert details passed as parameters.
2. Jenkins will use the parameters to determine which host to connect to and clean up the application's `tmp` directory.

We would need to:

1. Create a parameterized Jenkins job which can be triggered remotely.
2. Allow Jenkins to `ssh` into the application's host.
3. Setup a webhook receiver in Prometheus which calls the Jenkins job when a certain alert is fired.

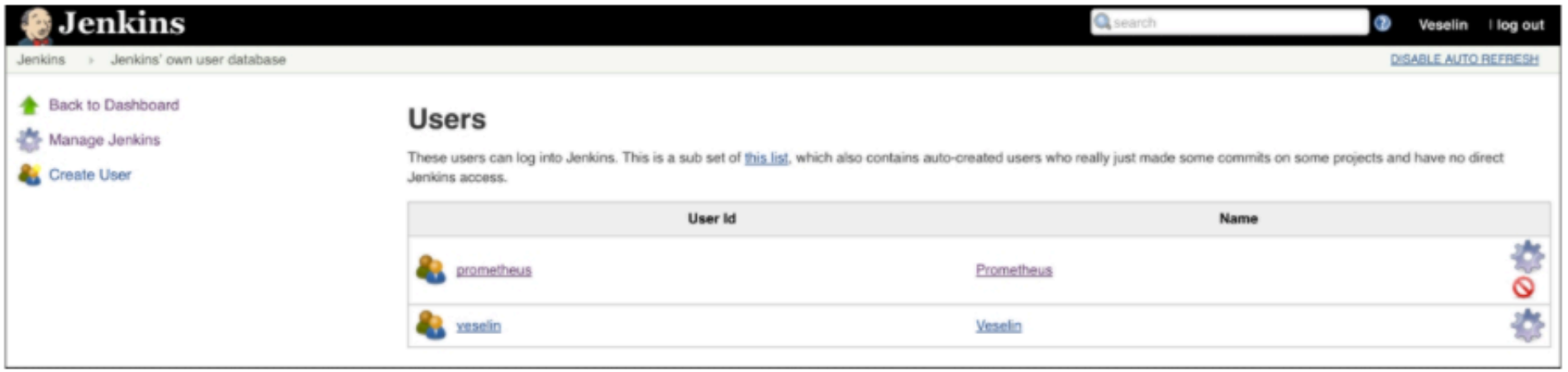
First a quick Jenkins installation onto our `promjenkins` node:

Copy

```
# yum install http://mirrors.jenkins-ci.org/redhat-stable/
      jenkins-2.7.1-1.1.noarch.rpm
# service jenkins start
```

`TCP: 8080` needs to be open, then you should be able to reach the Jenkins service at `http://$public_IP_of_promjenkins_node:8080` .

Under **Manage Jenkins** | **Manage Users** create an account for Prometheus:

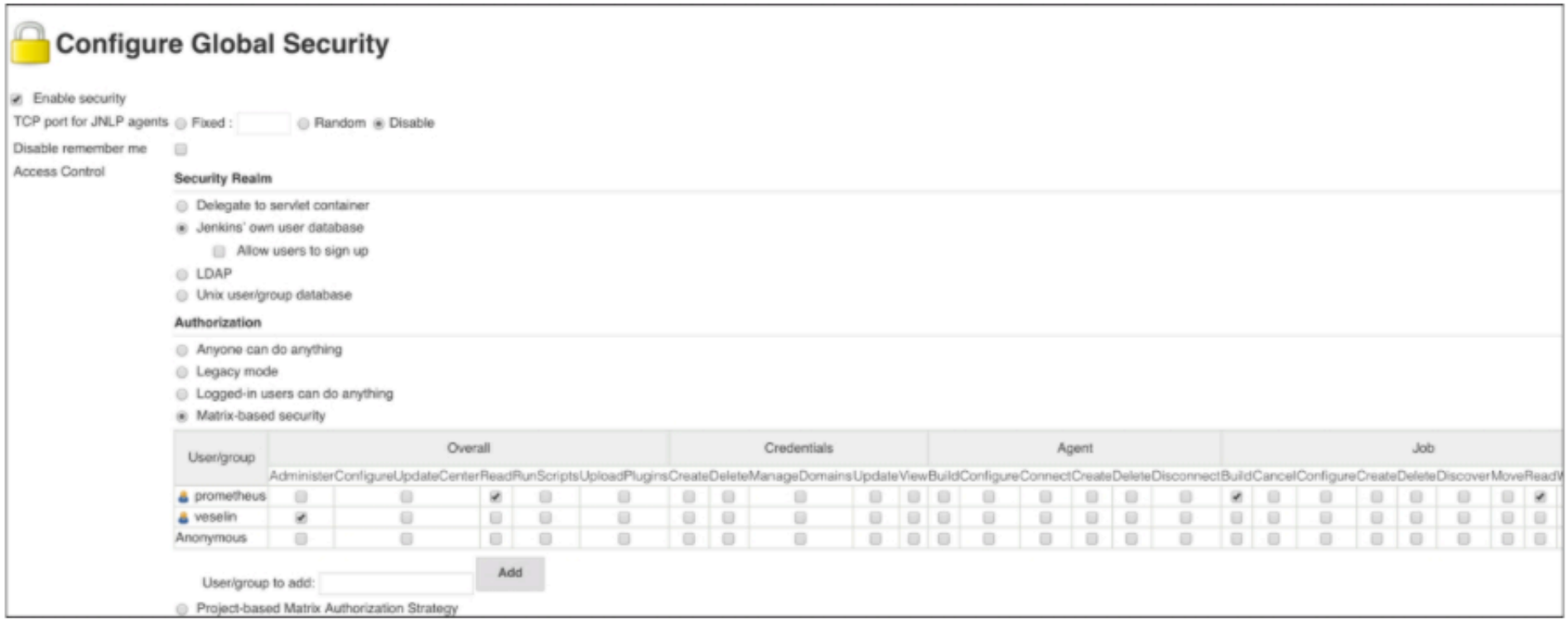


Then, under **Manage Jenkins** | **Configure Global Security**, select Jenkins' own user database and **Matrix-based Security**, then add both accounts.

Tip

Untick **Prevent Cross Site Request Forgery exploits** if you find that it causes issues when making `curl` request to Jenkins.

Grant yourself **Overall Administer rights** and **Prometheus Overall Read** plus **Job Build/Read**:



To be able to ssh into the app (web server) nodes we need a key for the Jenkins user:

Copy

```
# su - -s /bin/bash jenkins
$ ssh-keygen -trsa -b4096
Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/jenkins/.ssh/id_rsa):
Created directory '/var/lib/jenkins/.ssh'
...
```

While we are here, let us create an ssh config file for the Jenkins user (`~/.ssh/config`) containing:

Copy

```
Host 10.0.*
   StrictHostKeyChecking no
   UserKnownHostsFile=/dev/null
   User ec2-user
```

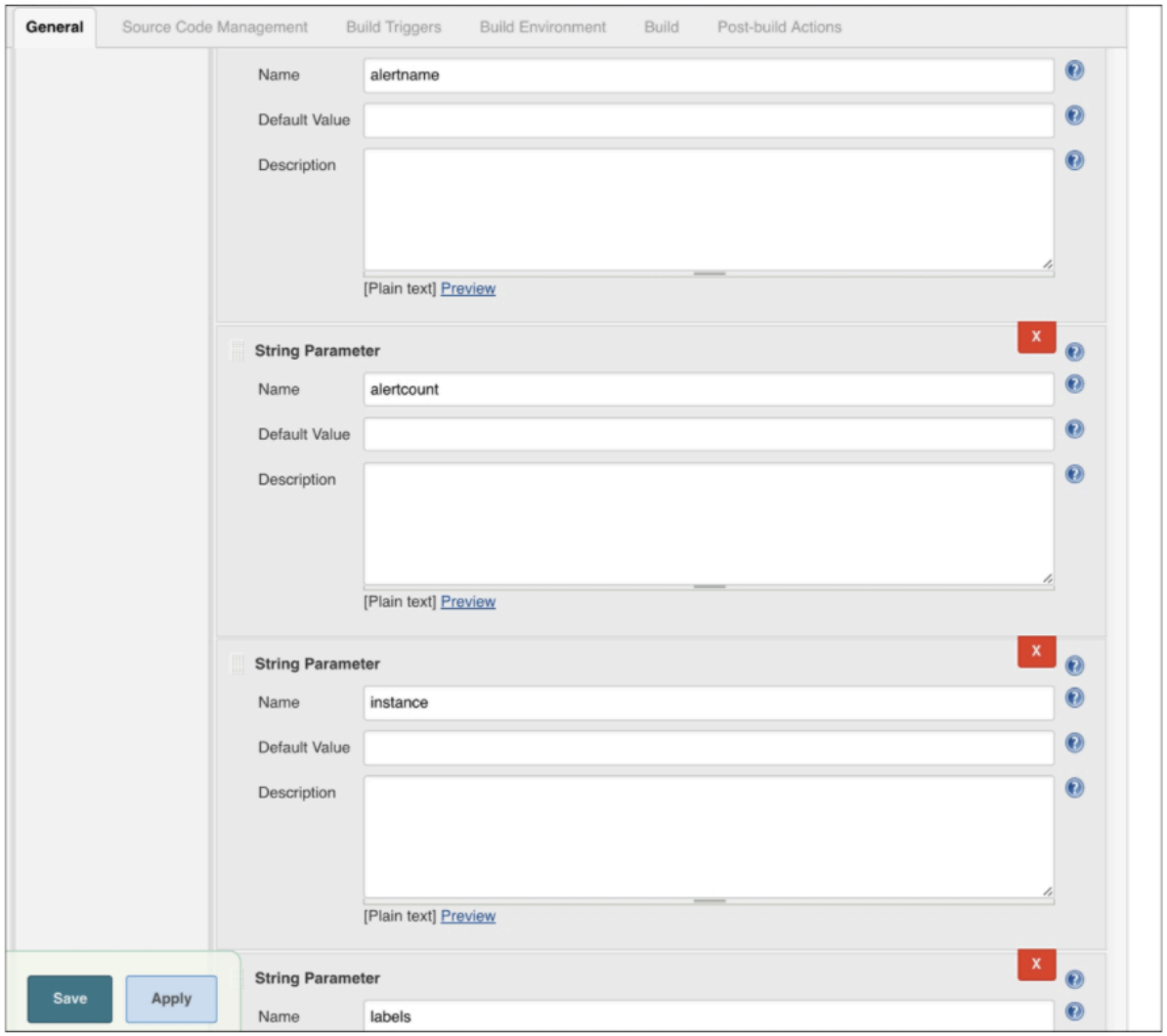
This is to allow our non-interactive jobs to ssh to instances for the first time.

We also need to take the generated public key and add it to the Auto Scale Group user data , so that it gets onto our web server instances. We will be using the standard (Amzn-Linux) ec2-user account to connect:

Copy

```
...
# Add Jenkins's key
cat << EOF >> /home/ec2-user/.ssh/authorized_keys
{{JENKINS_PUB_KEY_GOES_HERE}}
EOF
```

Now let us create the Jenkins job (freestyle project) with a few parameters:



We will discuss those four parameters (`alertname` , `alertcount` , `instance` , `labels`) later. In the **Build** section, select **Execute shell** and enter `exit 0` as a placeholder until we are ready to configure the job further. **Save** and let's get back to Prometheus.

As we mentioned earlier, we will be using the webhook receiver to trigger the Jenkins job. While the receiver allows us to set a URL to call, it does not seem to allow for any parameters to be included. To accomplish this, we will use a small helper application called **prometheus-am-executor** (ref: https://github.com/imgix/prometheus-am-executor).

The executor sits between the Alertmanager and an arbitrary executable. It receives the webhook call from the Alertmanager and runs the executable, passing a list of alert variables to it. In our case, we will be executing a shell script which processes those variables and constructs a `curl` call in the format that Jenkins expects.

Let us install the helper app alongside Prometheus and the Alertmanager:

Copy

```
# yum -y install golang
# mkdir /opt/prometheus/executor && export GOPATH=$_
# go get github.com/imgix/prometheus-am-executor
```

On success, you should have a binary in `/opt/prometheus/executor/bin` . Now the script (executable) that we mentioned:

Copy

```
#!/bin/bash

if [[ "$AMX_STATUS" != "firing" ]]; then
  exit 0
fi

main() {
  for i in $(seq 1 "$AMX_ALERT_LEN"); do
    ALERT_NAME=AMX_ALERT_${i}_LABEL_alertname
    INSTANCE=AMX_ALERT_${i}_LABEL_instance
    LABELS=$(set|egrep "^AMX_ALERT_${i}_LABEL_"|tr '\n' ' '|base64 -w0)
    PAYLOAD="{'parameter': [{'name':'alertcount', 'value':'${i}'}, {'name':'alertname', 'value':'${!ALERT_N
    curl -s -X POST http://localhost:8080/job/prometheus_webhook/build --user 'prometheus:password' --data-
  done
  wait
}

main "$@"
```

In essence we are constructing an HTTP call to our Jenkins job URL at `http://localhost:8080/job/prometheus_webhook/build` passing the `alertcount`, `alertname`, `instance` and `labels` parameters. All values come from the AMX environment variables which the prometheus-am-executor exposes (ref: https://github.com/imgix/prometheus-am-executor).

Now we need to reconfigure the Alertmanager to use webhooks:

```
global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'alertmanager@example.org'

route:
  group_by: ['alertname', 'cluster', 'service']
  group_wait: 10s
  group_interval: 30s
  repeat_interval: 1m
  receiver: team-X-mails

  routes:
  - receiver: 'jenkins-webhook'
    match:
      alertname: "High_disk_space_usage"

receivers:
- name: 'team-X-mails'
  e-mail_configs:
  - to: 'veselin+testprom@kantsev.com'
    require_tls: false
    send_resolved: true

- name: 'jenkins-webhook'
  webhook_configs:
  - url: http://localhost:8888
```

So, we have added a new sub-route which would match on `alertname` : `High_disk_space_usage` and use the `jenkins-webhook` receiver.

Reload Alertmanager and let us start the executor. Assuming that the `executor.sh` has been placed in `/opt/prometheus/executor` :

Copy

```
# cd /opt/prometheus/executor
# ./bin/prometheus-am-executor -l ':8888' ./executor.sh
2016/10/16 17:57:36 Listening on :8888 and running [./executor.sh]
```

We have the executor running (port `8888`) and ready to accept requests from the Alertmanager.

Before triggering any test alerts, let's go back to our Jenkins job. You are now familiar with the parameters it expects and the ones that we pass via the `webhook` | `executor` | `jenkins` setup that we have, so we can replace the contents of the placeholder **Build** step with this shell script:

Copy

```
echo "alertname: ${alertname}"
echo "alertcount: ${alertcount}"
echo "instance: ${instance}"

export $(echo ${labels}|base64 -d)

NODE=$(echo ${instance}|cut -d: -f1)
LABEL_DIR=AMX_ALERT_${alertcount}_LABEL_path
APP_DIR='/opt/myapp/tmp'

if [ ${!LABEL_DIR} == ${APP_DIR} ];then
ssh ${NODE} "sudo rm -f ${APP_DIR}/*.tmp"
fi
```

To test all of this, we need to ssh into one of the ASG (web server) instances which Prometheus is monitoring and setup a pretend App temporary folder like so:

Copy

```
# dd if=/dev/zero of=/tmp/dd.out bs=1M count=256
# mkfs.ext4 /tmp/dd.out
# mkdir -p /opt/myapp/tmp
# mount -oloop /tmp/dd.out /opt/myapp/tmp/
```

This should give us a small filesystem to play with. Next, we fill it up:

Copy

```
# dd if=/dev/zero of=/opt/myapp/tmp/dd.tmp bs=1M count=196
```

This is way over the 20% we have set in the `High_disk_space_usage` and should trigger it. In turn the executor should call Jenkins and run our job:

Console Output

```
Started by user Prometheus
Building in workspace /var/lib/jenkins/workspace/prometheus_webhook
[prometheus_webhook] $ /bin/sh -xe /tmp/hudson2570330293061564239.sh
+ echo 'alertname: High_disk_space_usage'
alertname: High_disk_space_usage
+ echo 'alertcount: 1'
alertcount: 1
+ echo 'instance: 10.0.4.134:9126'
instance: 10.0.4.134:9126
++ base64 -d
++ echo
QU1YX0FMRVJUXzFfTEFCRUxfYWxlcnRuYW1lPUhpZ2hfZGlza19zcGFjZV91c2FnZSBBTVhfQUxFUlRfMV9MQUJFTF9mc3R5cGU9ZXh0NCBBTVhfQUxFUlRfMV9MQUJFTF9ob3
N0PWlwLTEwLTAtNC0xMzQgQU1YX0FMRVJUXzFfTEFCRUxfaW5zdGFuY2U9MTAuMC40LjEzNDo5MTI2IEFNWF9BTEVSVF8xX0xBQkVMX2pvYj1lYzIgQU1YX0FMRVJUXzFfTEFC
RUxfbW9uaXRvcj1jb2RlbGFiLW1vbml0b3IgQU1YX0FMRVJUXzFfTEFCRUxfcGF0aD0vb3B0L215YXBwL3RtcCA=
+ export AMX_ALERT_1_LABEL_alertname=High_disk_space_usage AMX_ALERT_1_LABEL_fstype=ext4 AMX_ALERT_1_LABEL_host=ip-10-0-4-134
AMX_ALERT_1_LABEL_instance=10.0.4.134:9126 AMX_ALERT_1_LABEL_job=ec2 AMX_ALERT_1_LABEL_monitor=codelab-monitor
AMX_ALERT_1_LABEL_path=/opt/myapp/tmp
+ AMX_ALERT_1_LABEL_alertname=High_disk_space_usage
+ AMX_ALERT_1_LABEL_fstype=ext4
+ AMX_ALERT_1_LABEL_host=ip-10-0-4-134
+ AMX_ALERT_1_LABEL_instance=10.0.4.134:9126
+ AMX_ALERT_1_LABEL_job=ec2
+ AMX_ALERT_1_LABEL_monitor=codelab-monitor
+ AMX_ALERT_1_LABEL_path=/opt/myapp/tmp
++ cut -d: -f1
++ echo 10.0.4.134:9126
+ NODE=10.0.4.134
+ LABEL_DIR=AMX_ALERT_1_LABEL_path
+ APP_DIR=/opt/myapp/tmp
+ '[' /opt/myapp/tmp == /opt/myapp/tmp ']'
+ ssh 10.0.4.134 'sudo rm -f /opt/myapp/tmp/*.tmp'
Warning: Permanently added '10.0.4.134' (ECDSA) to the list of known hosts.
Finished: SUCCESS
```

We can see Jenkins connecting to the affected instance over SSH, then clearing our fake application `tmp` directory.

It is important to note that while we allow ourselves root access for the purpose of this example, in any other circumstances you would either ensure that Jenkins could handle the given `tmp` directory as a non-privileged user, or if you would absolutely have to use `sudo` and then limit the commands and command line arguments that can be used.