



# Linux Academy

## DevOps Monitoring

### *Study Guide*

Elle Krout

[elle@linuxacademy.com](mailto:elle@linuxacademy.com)

April 4, 2019

# Contents

Welcome to the Course! . . . . .	1
Deploying the Demo Application . . . . .	1
Application Overview . . . . .	1
Using Your Own Environment (Optional) . . . . .	2
Prometheus Setup . . . . .	2
Install Prometheus . . . . .	3
Alertmanager Setup . . . . .	6
Install Alertmanager . . . . .	6
Grafana . . . . .	8
Grafana Install . . . . .	8
Add a Data Source . . . . .	9
Add a Dashboard . . . . .	9
Push or Pull . . . . .	9
Which to Choose . . . . .	10
When to Pull . . . . .	10
When to Push . . . . .	11
Still Uncertain? . . . . .	11
Patterns and Antipatterns . . . . .	12
Thinking It's About the Tools . . . . .	12
What About Building One? . . . . .	12
Falling into Cargo Cults . . . . .	12
Not Embracing Automation . . . . .	13
Leaving One Person In Charge . . . . .	13
Service Discovery . . . . .	14
But What About Automation? . . . . .	15

Consul . . . . .	15
Zookeeper . . . . .	16
Nerve . . . . .	16
Platform-Specific Service Discovery . . . . .	16
Using the Node Exporter . . . . .	16
Installing the Node Exporter . . . . .	17
CPU Metrics . . . . .	19
Memory Metrics . . . . .	21
node_memory_MemTotal_bytes . . . . .	22
node_memory_MemFree_bytes . . . . .	22
node_memory_MemAvailable_bytes . . . . .	22
Disk Metrics . . . . .	23
File System Metrics . . . . .	24
Network Metrics . . . . .	26
Load Metrics . . . . .	26
Using cAdvisor to Monitor Containers . . . . .	27
Using the prom-client . . . . .	28
Counters . . . . .	30
Gauges . . . . .	32
Histograms and Summaries . . . . .	33
Redeploy the Application . . . . .	35
Rules . . . . .	36
For . . . . .	39
Annotations . . . . .	40
Labels . . . . .	41
Preparing our Receiver . . . . .	41

Using Alertmanager . . . . .	42
Silencing Alerts . . . . .	45
Adding a Dashboard . . . . .	45
Singlestat Panels . . . . .	47
Graph Panels . . . . .	47
Altering the Dashboard . . . . .	48
Suggested Dashboards . . . . .	48
Building a Panel . . . . .	48
Congratulations! . . . . .	50

# Welcome to the Course!

Any system benefits from monitoring, whether it be tracking CPU usage on the infrastructure, gathering custom metrics regarding application usage, or, ideally, both. After all, while we often put a lot of thinking into the building of our infrastructure and applications, we're going to spend more time maintaining it past that initial creation period — and monitoring is how we do that.

In this course, we'll be using Prometheus, Alertmanager, and Grafana to explore monitoring concepts and methodologies by getting hands-on with all three on both an application and infrastructure level. There will be plenty of hands-on labs, flash cards, and videos to help you on your monitoring journey, so let's get started!

## Deploying the Demo Application

### Application Overview

Before we get into the details of monitoring, we need an environment to monitor and monitoring tools to do the actual measuring of metrics. Since this course is *concept-focused* and not tool-focused, this means we're going to set up our environment with everything right from the start and have it running as we work. This is especially useful since we do want to feed out monitoring tool metrics, after all!

But before we download any monitoring tools, let's take a look at our existing environment. In the Cloud Playground, look at the server image for this course: DevOps Monitoring Deep Dive. Deploy it using whatever size you wish, then meet me back here when you're logged in.

Once logged in, let's move into the directory called `forethought` in the home directory of our default user, the `cloud_user`. Let's list the contents:

```
ls
```

Of these files, there's only a few we have to concern ourselves with: The `index.ejs` file, and the files under the `public` and `views` folder. These are the parts of the application that we can change — and will change, when it comes to monitoring this application. Until then, however, we just want to deploy it using Docker.

A Docker image has already been created:

```
docker image list
```

Called `forethought`, this image contains everything Docker needs to automatically start our application. The `Dockerfile` it was based on is provided if you'd like to take a look, but all we have to do to start the application is run:

```
docker run --name ft-app -p 80:8080 -d forefront
```

Then check that the container launched with:

```
docker ps
```

The `-p 80:8080` line in the `docker run` command mapped port 8080 on our container to port 80 on our

server, as well, which means we can navigate to either the public IP or provided URL to our server to check out the application itself.

As we can now see, Forethought is just a simple to-do application! We can add in some tasks, go on to remove them, and see a backlog of our completed items. Eventually, we're going to want metrics on how many tasks are added and completed and at what rates, but to do that, we need to set up our monitoring solutions first.

## Using Your Own Environment (Optional)

If you'd rather deploy everything out on your own environment, whatever that may be, you can download the application at the following GitHub repository, then follow the provided steps. We're using Ubuntu 18.04 in the provided course image and recommend you do the same, although these steps can be adapted for other Linux distros if needed.

### 1. Install Docker:

```
sudo apt-get install apt-transport-https ca-certificates \
curl gnupg2 software-properties-common`

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add

sudo apt-key fingerprint 0EBFCD88

sudo add-apt-repository "deb [arch=amd64] \
https://download.docker.com/linux/ubuntu bionic stable"

sudo apt-get install docker-ce
```

### 2. Add sudo-less Docker:

```
sudo usermod -aG docker USERNAME
```

Then refresh your session.

### 3. Install Node.js and npm:

```
curl -sL https://deb.nodesource.com/setup_10.x -o nodesource_setup.sh

sudo apt-get install nodejs

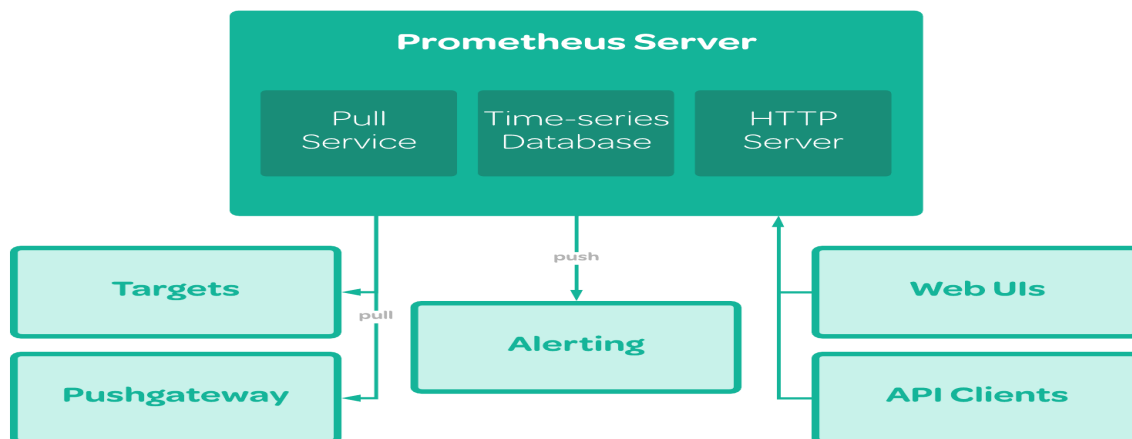
sudo apt-get install build-essential
```

### 4. Pull down the contents of the application in your user's home directory:

## Prometheus Setup

We now have the environment we want to monitor: Our application in a container that's working as a stand-in for a separate host, and our Cloud Playground host itself. We just need a monitoring solution.

In this case, we're going to use Prometheus, although Prometheus is just one of many monitoring solutions. We chose Prometheus, however, because it's the one most frequently referenced when it comes to approaching monitoring from a DevOps space, primarily due to its active scraping capabilities, native graphing and querying language, and time-series based alerting. It also has day-to-day features such as grouping, routing, and deduplication that can ease some frustration one may have in other monitoring platforms such as Nagios. And it has the benefit of being based on Google's BorgMon, referenced in the Google SRE book from which much of the foundation of monitoring within respect to DevOps is built on.



Prometheus works by scraping metrics across our system. We'll get into how these metrics are provided later, but right now all we need to know is that they are available for Prometheus to access and pull into its system. We also have access to a *push* gateway for short-lived jobs that may not be caught by our pull metrics.

Prometheus itself is made up of what is called the "Prometheus server": Which is Prometheus itself, a provided time-series database that stores our pulled metrics, and an HTTP server that provides us with the web UI.

Finally, Prometheus is able to push notifications to other applications — most usually an alerting application such as Alertmanager.

## Install Prometheus

Since we're using a Linux-based server with systemd, we're going to take the time to set up Prometheus so it's daemonized and we can manage it with the `systemctl` command. Do note that we can also simply download Prometheus and run the script, or deploy Prometheus in a container (see our [Monitoring Kubernetes with Prometheus course](#)).

At this point, we *could* simply download and extract the file and run the provided script. But since we want to set Prometheus up so we can daemonize it, we're instead going to take the time to create an application user and directories to separate and store our configuration files and libraries.

Let's first create the new user:

```
sudo useradd --no-create-home --shell /bin/false prometheus
```

Then the new directories:

```
sudo mkdir /etc/prometheus
```

```
sudo mkdir /var/lib/prometheus
```

And set the ownership of /var/lib/prometheus to the prometheus user:

```
sudo chown prometheus:prometheus /var/lib/prometheus
```

We now want to pull in the Linux .tar.gz file from Prometheus's [downloads page](#):

```
cd /tmp/
```

```
wget https://github.com/prometheus/prometheus/\nreleases/download/v2.7.1/prometheus-2.7.1.linux-amd64.tar.gz
```

Extract the files:

```
tar -xvf prometheus-2.7.1.linux-amd64.tar.gz
```

Let's now move into the extracted folder and take a look at what's provided here:

```
cd prometheus-2.7.1.linux-amd64/
```

```
ls
```

console\_libraries, consoles, and prometheus.yml will be stored in our /etc/ directory and all affect the configuration of Prometheus. prometheus.yml specifically references the configuration of Prometheus itself, while the two directories allow us to alter the web console if we so desired. Alternatively, both the prometheus and promtool files are binaries that we need to add to one of our bin paths. NOTICE and LICENSE are simply information files for us.

Move the configuration files and directories into the /etc/ folder:

```
sudo mv console* /etc/prometheus
```

```
sudo mv prometheus.yml /etc/prometheus
```

We now want to make sure prometheus is the owner of these files:

```
sudo chown -R prometheus:prometheus /etc/prometheus
```

Let's repeat the process, but this time moving our binaries to the /usr/local/bin/ directory:

```
sudo mv prometheus /usr/local/bin/
```

```
sudo mv promtool /usr/local/bin/
```

```
sudo chown prometheus:prometheus /usr/local/bin/prometheus
```



```
sudo chown prometheus:prometheus /usr/local/bin/promtool
```

Finally, we need to set up a systemd service file, letting us use `systemctl` to manage the starting, stopping, and enabling of Prometheus. To do this, we need to create a file, `prometheus.service`, in the `/etc/systemd/system/` directory:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/systemd/system/prometheus.service
```

Now just copy the following into the file:

```
[Unit]
Description=Prometheus
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
    --config.file /etc/prometheus/prometheus.yml \
    --storage.tsdb.path /var/lib/prometheus/ \
    --web.console.templates=/etc/prometheus/consoles \
    --web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target
```

Save and exit, then reload systemd:

```
sudo systemctl daemon-reload
```

We can now go ahead and start Prometheus:

```
sudo systemctl start prometheus
```

And ensure it starts on boot:

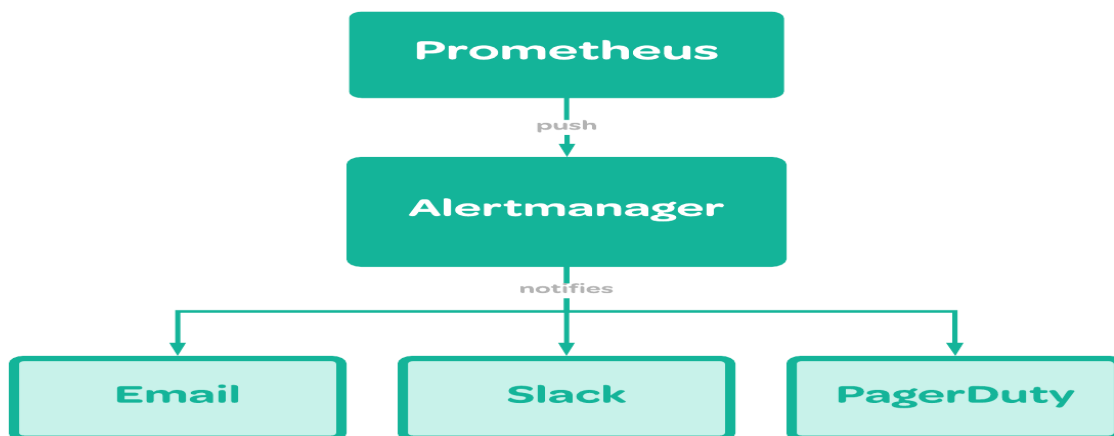
```
sudo systemctl enable prometheus
```

We can now confirm that Prometheus is running by going to the public IP address of our server, port 9090, in our web browser.

# Alertmanager Setup

Monitoring is never just monitoring. We don't decide to review trends in all our metric data without hoping to do something with these trends, and one of the things we can do with these trends is notice when something is wrong and make sure the right people are alerted. This is where the aptly-named concept of "alerting" comes in, along with the need for an alerting tool.

Prometheus does not ship with its own alerting tool, although one exists. Alertmanager can work outside of Prometheus, and is provided in its own set of binaries. Like with Prometheus itself, we can download these and simply run them, or we can set Alertmanager up to work with `systemctl` and with all its files in the expected spots.



Alertmanager does not just push notifications to our defined endpoints, however. It also supports grouping, inhibition, silences, and high availability, plus some advanced client behavior features for non-Prometheus setups.

However, since we're not using anything that requires the modification of the Alertmanager client, we can go ahead and start the setup process.

## Install Alertmanager

Like with our Prometheus setup, we want to create a new system user, named `alertmanager`, to run the Alertmanager itself:

```
sudo useradd --no-create-home --shell /bin/false alertmanager
```

We also want to create the `/etc/alertmanager` directory to store our files:

```
sudo mkdir /etc/alertmanager
```

Now, let's go ahead and pull the [Alertmanager download](https://github.com/prometheus/alertmanager/) into our `/tmp/` folder:

```
cd /tmp/
```

```
wget https://github.com/prometheus/alertmanager/
```

```
releases/download/v0.16.1/alertmanager-0.16.1.linux-amd64.tar.gz
```

And extract the files:

```
tar -xvf alertmanager-0.16.1.linux-amd64.tar.gz
```

```
cd alertmanager-0.16.1.linux-amd64/
```

```
ls
```

Just like with Prometheus, we're provided with two binaries, `alertmanager` and `amtool`; and a configuration file, `alertmanager.yml`. Move the two binaries into the `/usr/local/bin` directory and set the ownership to the `alertmanager` user:

```
sudo mv alertmanager /usr/local/bin/
```

```
sudo mv amtool /usr/local/bin/
```

```
sudo chown alertmanager:alertmanager /usr/local/bin/alertmanager
```

```
sudo chown alertmanager:alertmanager /usr/local/bin/amtool
```

Repeat the process, only this time moving the default `alertmanager.yml` file into the `/etc/alertmanager` directory:

```
sudo mv alertmanager.yml /etc/alertmanager/
```

```
sudo chown -R alertmanager:alertmanager /etc/alertmanager/
```

Now, let's create the `alertmanager.service` file in our `/etc/systemd/system` directory:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
$EDITOR /etc/systemd/system/alertmanager.service
```

```
[Unit]
```

```
Description=Alertmanager
```

```
Wants=network-online.target
```

```
After=network-online.target
```

```
[Service]
```

```
User=alertmanager
```

```
Group=alertmanager
```

```
Type=simple
```

```
WorkingDirectory=/etc/alertmanager/
```

```
ExecStart=/usr/local/bin/alertmanager \
    --config.file=/etc/alertmanager/alertmanager.yml
```

```
[Install]
```

```
WantedBy=multi-user.target
```

Save and exit.

Before we start the Alertmanager, we need to update our Prometheus configuration, as well. Stop the prometheus service:

```
sudo systemctl stop prometheus
```

And update the `/etc/prometheus/prometheus.yml` file, uncommenting the Alertmanager configuration and setting the target to `localhost:9093`:

```
sudo $EDITOR /etc/prometheus/prometheus.yml
```

```
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      - localhost:9093
```

Reload systemd and start both the prometheus and alertmanager services:

```
sudo systemctl daemon-reload
```

```
sudo systemctl start prometheus
```

```
sudo systemctl start alertmanager
```

Let's also make sure alertmanager starts at boot:

```
sudo systemctl enable alertmanager
```

To confirm that everything is working, navigate to port 9093 of your server in your web browser.

## Grafana

You may have already guessed that a lot of monitoring is about metrics, and while Prometheus provides us with a UI and the PromQL query language to look at that data, there are more robust solutions to visualizing our data. Grafana is one of these solutions, allowing us to create various graphs on our Grafana dashboards, not just from Prometheus but from any data source that might benefit from that data being visualized. You can even “mix and match” your data sources, comparing metrics across a variety of systems.

Grafana setup is also a lot simpler than Prometheus and Alertmanager.

## Grafana Install

To set up Grafana, all we have to do is add the required prerequisite package:

```
sudo apt-get install libfontconfig
```

Then pull down and install the .deb file from the [Grafana download page](#):

```
wget https://dl.grafana.com/oss/release/grafana_5.4.3_amd64.deb
```

```
sudo dpkg -i grafana_5.4.3_amd64.deb
```

The Grafana server is automatically started, so now all we have to do is make sure it starts on reboot:

```
sudo systemctl enable grafana-server
```

We now want to navigate to port 3000 on our server in the web browser. We'll be prompted for a username and password, both of which are `admin`. Change the password when prompted.

## Add a Data Source

The last thing we want to do before finishing with the basic setup of our deployment is add Prometheus as one of our data sources in Grafana. Click on the **Add data source** button on the home dashboard, then select **Prometheus**.

Set the **URL** to `http://localhost:9090`, then click **Save & Test**.

## Add a Dashboard

Return to **Home** from the left menu. We're prompted that the next step is to create a new dashboard; click on the **New dashboard** button.

A dashboard is automatically created. To change its settings, click on the gear icon to the upper right. Change the **Name** value to `Forethought`, then **Save**.

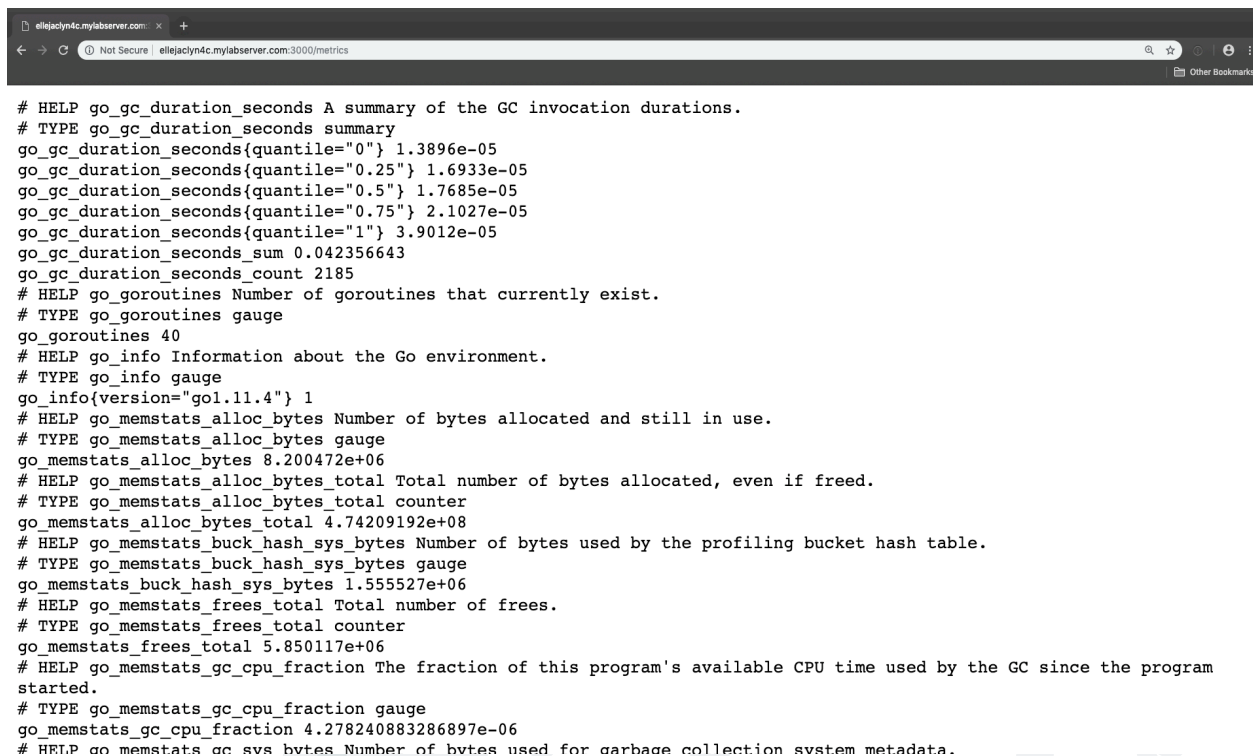
We're not yet going to add any charts — we need to address what types of metrics we're working with first — so we can go ahead and leave Grafana for the time being.

## Push or Pull

Although we've already set up our monitoring solution, when the time comes for you to choose what will be monitoring your actual systems, one of the first thing you'll want to consider is whether you'll benefit from a *push* monitoring solution, or a *pull* monitoring solution.

"Push" and "pull" specifically relates to *how* we're receiving our metrics. This means that on a pull system, like Prometheus, the system scrapes various endpoints that we expose and pulls that data into our system. For an example of this, we can actually use our Grafana setup, since Grafana provides us a metrics endpoint without any configuration at all. Simply navigate to

`YOURLABSERVER:3000/metrics`.



```
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 1.3896e-05
go_gc_duration_seconds{quantile="0.25"} 1.6933e-05
go_gc_duration_seconds{quantile="0.5"} 1.7685e-05
go_gc_duration_seconds{quantile="0.75"} 2.1027e-05
go_gc_duration_seconds{quantile="1"} 3.9012e-05
go_gc_duration_seconds_sum 0.042356643
go_gc_duration_seconds_count 2185
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 40
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.11.4"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 8.200472e+06
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 4.74209192e+08
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.555527e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 5.850117e+06
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 4.278240883286897e-06
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
```

As we can see here, we have lines and lines of text data provided to us at this endpoint. And while this data doesn't probably mean much to us humans, it means a lot to Prometheus.

So how does Prometheus receive this data? Well, as we mentioned, Prometheus *pulls* data in: After a set amount of time (every 15 seconds, by default), it connects to the endpoint and retrieves the data, putting the burden of gathering metrics on the monitoring system itself.

In contrast, push-based systems place the burden on the system *being* monitored. In other words, every set amount of time, the monitored system pushed the information up to the monitoring solution – think of it like a `git push` but with metric data. So if we were monitoring our Grafana setup with a push-based system, a service on that system would be providing Prometheus with metrics by initiating a connection *to* Prometheus, then sending the data that way.

Of course, we already noted that Prometheus *can* accept pushed metrics – or, rather, the Pushgateway can. Intended specifically for short-lived service jobs that might be missed during the scraping process, these jobs push their metrics to the Pushgateway, which Prometheus then scrapes as usual.

## Which to Choose

Asking someone knee-deep in monitoring whether to choose a push or pull system can be a contentious subject. Much like determining whether Vim is superior to Emacs, a lot of people have a lot of feelings about their monitoring solution. But the truth is both have their benefits and their drawbacks.

## When to Pull

Many people's first introduction to pull-based systems is Nagios, which is known for being hard to scale. This is often related to how Nagios spawns subprocesses when performing active checks; however, this

ignores that these checks *can* be run client-side, and also that not all pull-based systems are Nagios and thus won't have the same growing pains as Nagios does.

Concern also comes in when considering how “live” the checks are. In a push-based system, metrics are received by the monitoring system immediately; in contrast, if you're using an *event-driven* pull-based system, there can be a buffering process. In event-driven monitoring solutions, the system takes an event — such as an HTTP connection — and turns that into a metric, which often involved later processing. However, most pull systems are not event-driven — Prometheus certainly is not — and often these event-based systems perform different tasks than straight metrics monitoring, such as log aggregation.

Of course, that doesn't mean there's no drawbacks to pull-based monitoring. If you have a large infrastructure that spans across continents with elaborate firewalls and networking setups — anything that would prevent Prometheus from accessing the metrics endpoint — then a push-based system is probably for you.

## When to Push

The primary benefit of a push-based monitoring solution is it takes a lot of the taxing work — that is, pulling metrics — off the monitoring system and places it into the hands of the system being monitored. This means the monitoring application isn't sending hundreds or thousands (or more!) connections out, it just has to sit back and accept the data (which, of course, *does* mean receiving hundreds or thousands (or more!) connections).

Push also plays much nicer when you *do* need event-based monitoring. Instead of waiting the requisite 15 seconds to pull in data, it can be sent in real-time as the event happens. This allows for a faster collection of data, and prevents any missed jobs that might be especially short-lived — those same ones Prometheus had to create the Pushgateway to address.

And, of course, we can also go back to the large and varied infrastructure issue: If you *do* have a lot of firewalls, or a complicated infrastructure, push-based systems allow for greater compartmentalization and modularity. A pull-based system is, in many ways, a monolith: It's doing a lot of the work. Push systems offload much of this work to the systems sending the data.

## Still Uncertain?

Although this course uses a pull-based system, the information contained in it can be taken and used on many push-based systems, as well: Metrics, after all, don't particularly care *how* they end up in the monitoring system, so long as the data is there and we can learn from it.

If you're not certain what is best for your monitoring needs, I would suggest giving both a push and a pull system a try, and seeing which gives you the most trouble to set up in a replica of your existing infrastructure. Chances are, if you're having trouble in your dev environment, you'll encounter that in production, and you'll want to consider the alternative system.



# Patterns and Antipatterns

One of the biggest confusions regarding monitoring is just how open-ended it can be. And because of this, it's very easy to fall into certain "traps" regarding monitoring: Ways of doing things that seem like they make sense, but actually cause more trouble than they should. The following anti-patterns (and ways to combat them) are just a few of the issues we face when setting up a monitoring solution, but are some of the most common ones.

## Thinking It's About the Tools

You might be tempted, after taking this course, to just use Prometheus to monitor things because that's what we used. Why learn another application when you just got hands-on with one, right?

Well, this brings up one of the primary problems people have when they start monitoring: They try to fit their needs into the tools they have chosen, when in actuality they need to pick the tools that fit their needs. We discussed push versus pull monitoring in our last lesson, and that's just part of it. Maybe your system will benefit from push monitoring; maybe a pull-based system is good but Prometheus isn't quite what you need.

It's also easy to want to jump from tool to tool. Are you already using Nagios? Do you really need to jump ship and go to Prometheus or can you simply look at your current setup and think of a way to refine it? Are the problems you have with Nagios going to come with you to Prometheus? It's easy to blame the tool, but getting caught in a cycle of switching your monitoring every time you hit a roadblock will use up valuable time where you could be doing anything else and trusting that your monitoring system will alert you when things go wrong.

That said, don't be afraid to add tools that will benefit you, as well. Even if you end up with the exact setup of the course, you'll probably want to look into a log aggregation platform, such as setting up an ELK stack. Just be sure when you're adding a new tool you're not duplicating work (unless the plan is to abandon the old system for the new).

## What About Building One?

Seeing all the caveats about picking monitoring tools might make you pause and think, "Well, maybe I should build one!" But unless you work for a company that has the time and money to invest in in-house solutions at scale (like Google did with BorgMon), chances are you're best getting something out of the box and adding functionality to it as needed. It's easier to write an exporter for Prometheus than a whole new Prometheus, after all.

## Falling into Cargo Cults

It's easy to look at how everyone else does monitoring and think "Well, I'll just do that," but "just doing that" is rarely the best way to meet your company's individual needs. Are you alerting for CPU because "everyone else does it" but when the alerts come in you don't know what the actual problem is? If so, chances are you're monitoring for the wrong things for the wrong reasons: Namely, because everyone else is doing it, and the time wasn't taken to figure out what you actually need.



Which isn't to say we don't want to monitor our CPU and memory, but we need to know *why* and we need to ensure our alerting is set up to not just tell us that something is wrong, but to alert us about the correct issue. Does our CPU go wild when we have too many things processing at once? Then we want to monitor how many processes our application is generating and alert when it hits *that* threshold, not the ambiguous CPU threshold, even though it seems most obvious to monitor and alert on the CPU.

If you think you're setting up part of your monitoring solution just to follow a trend, then it's important to ask "why?" If the answer is "because I read about it in the Google SRE book and it sounds cool" then you probably shouldn't be spending time on it. Of course, that doesn't mean you *can't* or *shouldn't* take inspiration from others' successes, just be sure that it's something you'll actually benefit from.

## Not Embracing Automation

We'll talk about this a little more in a future topic, but once your monitoring solution is actually set up, the work is still not done. Other projects, additional servers, new applications, and more will all have to be added to the monitoring landscape as they are created. This process should be automated using some kind of service discovery system and self-enrollment (if using a push-based system). Otherwise, there's substantial chance that things will go unmonitored as we wait for someone to enroll any new endpoints that need monitoring.

Additionally, we also want to consider what we're consistently being alerted on. Generally, when we do any kind of incident response, a *runbook* is created. A runbook is an information document noting the problem and the steps that took place to fix the problem. So if you happen to have the same issue again and again and the actions taken by incident response is the same: Instead of alerting someone, automate the solution! After all, it's not uncommon to see setups where new virtual machines are deployed when a certain CPU limit is reached; why shouldn't we be applying that idea to all of our common incidents?

## Leaving One Person In Charge

Monitoring is not a one-person job. Every platform, every application, every container should be monitored in some way, and it should not be the job of one person to implement monitoring on everything. After all, is your systems person the best employee to write the monitoring parts of a Ruby application? Or is it your existing Ruby dev? Especially when things like service discovery are added to the mix, and the need for human intervention is eliminated when it comes to including a new service to our monitoring platform.

And because of this, it means that we *all* need to be monitoring-minded. It's not good enough to place the job of monitoring on one person or one team. If we're monitoring an application, dev should be thinking about how to best monitor what's going on when the application is being used; if we're monitoring a container or virtual machine, systems needs to consider what common problems we need to look out for given that container or machine's task (is it running something that takes up a lot of memory? Do you have file system space concerns?). After all, the person involved in a project from the start is going to have a better idea of what to look out for regarding that project than someone who would come in just to add monitoring.

# Service Discovery

Up until this point in the course, we have been using a lot of words for the same concept: Client, endpoint, service, target... All of these are just simple words for *something that our monitoring system is monitoring*. And we can see what targets we're monitoring on Prometheus by going to our Prometheus web UI, and navigating to the **Targets** page, located under the **Status** menu option. Right now, we're just monitoring Prometheus itself, but we're obviously going to want to add more.

Here is where service discovery comes in. Service discovery is simply the way Prometheus detects our metrics pages for any of our desired targets. If you remember a few lessons ago, we actually looked directly at the metrics page for our Grafana setup. Well, if we wanted Prometheus to discover that page itself, we would have to add that information to our Prometheus configuration.

Go ahead and open up `/etc/prometheus/prometheus.yml` now:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/prometheus/prometheus.yml
```

At the bottom of the configuration file, we have a section for `scrape_configs`, that currently only has information about the prometheus monitoring job:

```
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['localhost:9090']
```

To add a job for Grafana, we only have to mimic this for our Grafana setup:

```
- job_name: 'grafana'
  static_configs:
    - targets: ['localhost:3000']
```

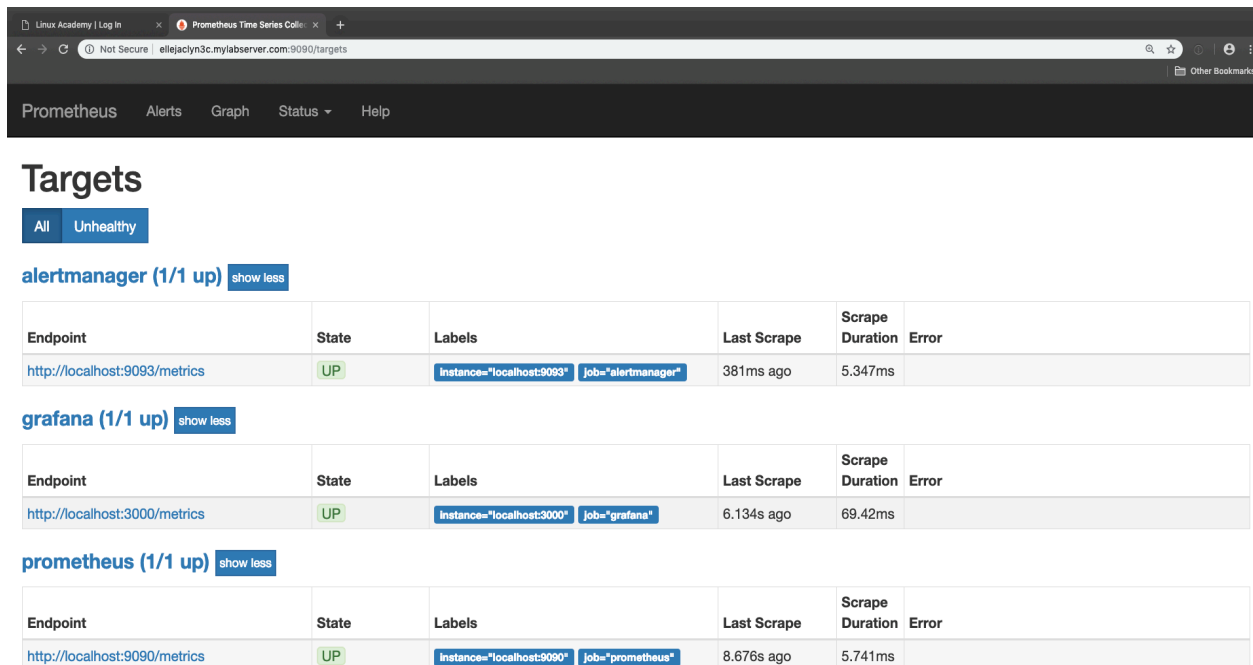
While we're here, let's also add Alertmanager:

```
- job_name: 'alertmanager'
  static_configs:
    - targets: ['localhost:9093']
```

Save and exit. Now, to get Prometheus to discover these new targets, all we have to do is restart our Prometheus service:

```
sudo systemctl restart prometheus
```

If we refresh the **Targets** page, we can now see that we have three targets up and running:



The screenshot shows the Prometheus web interface with the 'Targets' tab selected. There are three sections, each with a table of targets. Each section has a 'show less' link.

### Targets

**alertmanager (1/1 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9093/metrics	UP	instance="localhost:9093" job="alertmanager"	381ms ago	5.347ms	

**grafana (1/1 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:3000/metrics	UP	instance="localhost:3000" job="grafana"	6.134s ago	69.42ms	

**prometheus (1/1 up)** [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	8.676s ago	5.741ms	

## But What About Automation?

If you're fresh off our last lesson, you might be seeing all this above and wondering where automation fits in to this. After all, we already said that one of the most common anti-patterns is *not* automating the service discovery process. And while there are some ways to use DNS and regular expressions to automate service discovery in Prometheus, the best way to do it for most people is to add some kind of service discovery tool.

Since we only have a very limited amount of endpoints in our setup for this course, we're not going to add any of these tools to our current setup. That said, we will be taking the time to go over some popular ones, all of which are already supported by Prometheus and have their own `scrape_config` settings readily available.

Do note that although all of these services offer service discovery, they are not exclusively service discovery tools. If there's one tool you want to pay especially close attention to when choosing, it's this one.

## Consul

**Consul**, by HashiCorp, is a "service mesh solution" that offers service discovery among other related fea-

tures, such as performing health checks, and segmentation. Consul can work across multiple data centers to discovery and register services and clients, which we can then use in other programs such as Prometheus. Of all the options listed, Consul is the most “pure” form of service discovery, working across whatever your infra setup is and serving only functions that relate directly to high-level service management.

## Zookeeper

Apache’s [Zookeeper](#) works as both a central hub for distributed systems and a key-value store. Originally part of the Hadoop projects, Zookeeper is considered a project in its own right. Zookeeper helps us name, group, and synchronize services across large infrastructures, however has enough limitations that it can be used in conjunction with...

## Nerve

Nerve, by AirBnB, combines the powers of Synapse and Zookeeper to automatically enroll services into Zookeeper, further automating the process out. Nerve also keeps track of the status of the machines it watches and reports information back to Zookeeper’s key-value store. This, along with Zookeeper, allows for very robust configuration and service tracking, although it is a more involved system than Consul.

## Platform-Specific Service Discovery

You may also find that there are advanced service discovery options in your existing platform setup. For example, AWS, Azure, and Google Cloud Platform all offer their own options for service discovery, while container orchestration platforms like Kubernetes and Marathon can also perform contained-based service discovery — and all of these platforms are already supported in Prometheus, we just need to update the configuration file once to add the service discovery endpoint. If you’re working on a more modest infrastructure, this is probably the best place to start, especially if you don’t have the time or overhead to set up an entirely new solution. You’ll be getting the same amount of automation for less of the work.

## Using the Node Exporter

If we currently view our Prometheus web UI and search for any metrics in our expression editor, we can see that we’re currently quite limited in scope: We can see metrics for our monitoring systems but nothing else at all.

---

☐ Enable query history

Expression (press Shift+Enter for newlines)

Execute

- insert metric at cursor -

But a monitoring system that only monitors itself is no good: We need to start pulling in data. And some of the simplest data we can pull in while we get used to monitoring is host data — things like our CPU, memory, and disk space. In Prometheus, we’re going to gather this data by using the *node*

*exporter*, although other monitoring systems will have different means of gathering this data; *collectd*, for example, is particularly popular with push-based solutions.

## Installing the Node Exporter

To install the Node Exporter, we'll be working the same way we did for Prometheus and Alertmanager: By downloading the needed files, adding a system user, and creating a systemd service file so we can start and stop the exporter as needed.

Let's first create that system user:

```
sudo useradd --no-create-home --shell /bin/false node_exporter
```

Then download and extract the `.tar.gz` file from [Prometheus's download page](#):

```
cd /tmp/

wget https://github.com/prometheus/node_exporter\
/releases/download/v0.17.0/node_exporter-0.17.0.linux-amd64.tar.gz

tar -xvf /releases/download/v0.17.0/node_exporter-0.17.0.linux-amd64.tar.gz
```

Move into the new directory:

```
cd node_exporter-0.17.0.linux-amd64/
```

Here we have only one file we need to work with: the `node_exporter` binary. Let's move it to the `/usr/local/bin` directory:

```
sudo mv node_exporter /usr/local/bin/
```

And set the appropriate ownership:

```
sudo chown node_exporter:node_exporter /usr/local/bin/node_exporter
```

Next, we need to create the systemd file for the service:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
$ sudo $EDITOR /etc/systemd/system/node_exporter.service
```

```
[Unit]
Description=Node Exporter
After=network.target
```

```
[Service]
User=node_exporter
```

```
Group=node_exporter
Type=simple
ExecStart=/usr/local/bin/node_exporter
```

```
[Install]
WantedBy=multi-user.target
```

We can now reload systemd and start the node\_exporter:

```
sudo systemctl daemon-reload

sudo systemctl start node_exporter
```

Finally, we can add the node\_exporter to our endpoints so Prometheus can scrape the data:

**Note:** Replace \$EDITOR with the text editor you prefer to use (vi, vim, nano, etc.)

```
$ sudo $EDITOR /etc/prometheus/prometheus.yml

- job_name: 'node_exporter'
  static_configs:
    - targets: ['localhost:9100']

$ sudo systemctl restart prometheus
```

Now, if we return to Prometheus and refresh, when we try to search for a metric, various system metrics are now available. To see for yourself, search memory and view the node\_memory\_MemFree\_bytes metric, which provides data on the amount of free memory we have. If we switch the time series, we can see some changes, but what we really want to do is watch some bigger spikes in the graph; for this we can use a tool called stress:

```
sudo apt-get install stress
```

And force some load on our RAM:

```
stress -m 2
```

Wait for a minute, then refresh the Prometheus graph; there will be a noticeable difference in activity.



## CPU Metrics

Before we begin this lesson, it is suggested you log in to your server and run a stress test against your CPU. To do this, run:

```
stress -c 5
```

With the Node Exporter now set up, we have access to a number of system metrics on our server. However, you may recall from previous lessons that I warned system metrics aren't always the best at determining a problem — that doesn't mean we shouldn't record or understand them, though! While in most cases we should not be *alerting* on these metrics, being able to navigate and understand the metrics available for each part of our infrastructure is integral for us to use our data to make good decisions about our platform.

We'll start with one of the core components of your functioning server: The CPU.

Prometheus and the majority of monitoring programs receive their data about CPU from one source: the `/proc/stat` file on the host itself. Metrics involving the CPU are prefixed with `node_cpu`. Let's check one out now by searching for and selecting `node_cpu_seconds_total` in our expression editor on the web UI. Switch to the **Console** view.

`node_cpu_seconds_total` is what is known as a *counter* — that is, it keeps a running total of the amount of time the CPU is in each defined state, in seconds. As such, you'll only see these numbers going up. This metric measures the amount of time of each of the following CPU states:

- `idle`: Time the CPU is doing nothing
- `iowait`: Time the CPU is waiting for I/O
- `irq`: Time spent on fixing interrupts
- `nice`: Time spent on user-level processed with a positive nice value
- `softirq`: Time spent fixing interrupted
- `steal`: In running a virtual machine, time spent with other VMs "stealing" CPU

- **guest:** On servers that *host* VMs, this value will be *guest* and contain the amount of CPU usage of the VMs hosted
- **system:** Time spent in the kernel
- **user:** Time spent in userland

But this information isn't too useful on it's own. On our own demo servers, we can see that we're mostly running idle, but beyond that, when we view the graph of this data, we can see that it's not very helpful at all. This is where some math comes in.

Prometheus uses a language called PromQL that lets us run queries against this metric data, performing tasks like calculating averages, means, and other mathematical functions. So let's say we want to see what *percentage* of CPU is in each state at a certain time frame. In this case, we would use either the *rate* or *irate* querying function. *rate* calculates the per-second average change of the time series in the range; *irate* performs the same function, but is intended for particularly fast-moving counters. That probably sounds a little confusing, so let's see it in action. Input the following into the expression editor:

```
irate(node_cpu_seconds_total[30s]) * 100
```

Move to the graph page, then change the limit on the graph to 30m, and we can now see how much more interesting our graph looks!



Your own graph will vary from the one in the image, but here we can see the *idle* mode drops as the time the CPU spent working on our user-related tasks spikes. And since I ran *stress* with a particularly high CPU rate, we can also see that after a bit, it begins to *steal*. We're also using *rate* here, which, with the 30s interval specified, does not look that much different than if we were to run it with *irate*, but should we move the 30s up to 1m or even higher, you can notice how *rate*, in the case of our CPU cycles, provides less specific data.

We can, of course, get more specific than just general percentages, as well. If we wanted to specifically calculate the percent of time the CPU is in a *steal* cycle, for example, we can do some tweaking to our above expression and run:

```
irate(node_cpu_seconds_total{mode="steal"}[1m]) * 100
```



Note the addition of the `{mode="steal"}` to the expression, which lets us clarify that we only want data about the amount of time we're running in `steal`.



Additionally, although we are working with only one server, I do want to note how we can calculate the average idle time across multiple instances. To do that we should just run:

```
avg by (instance) (irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100
```

It should also be noted that there is one other CPU metric available to us: `node_cpu_guest_seconds_total`, specifically geared toward measuring the CPU usage of guests. Since this isn't pertinent to our setup, the data here is pretty unremarkable, but in infrastructures where this is valid, we would manage it that same as we did above.

Running the `stress` command still? Now would be a great time to end that process.

## Memory Metrics

Before we begin this lesson, it is suggested you log in to your server and run a stress test against your memory. To do this, run:

```
stress -m 1
```

If our servers are struggling and the issue isn't CPU, there's a good chance it's going to be memory. Like CPU, memory is something we want to track but not necessarily alert on in our monitoring solution. With memory in particular, the core questions we want to ask are: How much of it is free? And how much of it is used?

Of course, we can go a little deeper here, as well. There are three "types" of memory, much like there were multiple modes for our CPU. These are *shared*, *cached*, and *buffering*.

*Shared* memory is memory that can be accessed by multiple programs simultaneously, whereas *cached* and *buffering* memory store recently-accessed data, with buffers specifically storing the metadata and caches storing the actual contents of the accessed file (remember, when I say "file" here, that everything in Linux is a file).

Memory in buffers and caches are important because this memory *can be reclaimed*. So if we're experiencing issues due to low memory and yet most of it is being taken up by buffers and caches, a simple fix would be to clear those buffers/caches.

Prometheus, like most monitoring systems, gathers metrics data for memory from `/proc/meminfo`, which we can also view with the `free` command on our terminal. In Prometheus itself, this data uses the `node_memory` prefix, and compared to our CPU data, there are a large amount of expressions to choose from; however, most of us will only need to consider a few of these, specifically:

## node\_memory\_MemTotal\_bytes

`node_memory_MemTotal_bytes` defines the amount of memory available on a server. Note that this is not *free* memory, but rather how much memory the system has in general. For example, if you're using a micro-sized playground server, the number here is most likely around 1000000000. It also means that, unless we go in and actually add or allocate more memory for this server, the graph will look like a simple, straight line.

That said, this metric is useful if we ever need to calculate something against our full memory. And if you're using Prometheus to monitor any servers on older kernels, you absolutely *will* have to use it. Older kernel version do not support metrics that record the amount of available memory, so to gain that information, we will have to use a combination of this and...

## node\_memory\_MemFree\_bytes

I suggest stopping the `stress` process now.

`node_memory_MemFree_bytes` is the amount of memory "free" at the given points in time, in bytes. However, this does *not* include the amount of memory we can reclaim from caches and buffers, meaning that it is technically not that full amount of memory *available*, but we'll talk on this later. To calculate the amount of in-use memory, subtract this metric from the `node_memory_MemTotal_bytes` metric:

```
node_memory_MemTotal_bytes - node_memory_MemFree_bytes
```

We could also turn this into a percentage, if that helps us better understand the state of our system:

```
((node_memory_MemTotal_bytes - node_memory_MemFree_bytes)/node_memory_MemTotal_bytes) * 100
```

## node\_memory\_MemAvailable\_bytes

While `node_memory_MemFree_bytes` outputs the amount of memory actually *free* during the time series, `node_memory_MemAvailable_bytes` provides the amount free, alongside the amount we can reclaim from the buffers (`node_memory_Buffers_bytes`) and caches (`node_memory_Cached_bytes`), giving us a true availability amount.

However, there are some things we need to know about this metric: That is, it is only available on systems using kernel version 3.14 and newer. So any old or legacy systems you're still monitoring and maintaining will need to calculate this metric by hand, by adding the `MemFree`, `Buffers`, and `Cached` metrics:

```
node_memory_MemFree_bytes + node_memory_Buffers_bytes + node_memory_Cached_bytes
```

---

All of these expressions capture our metrics in *gauges*. Unlike our CPU, which kept a running count of how long our CPU spent in each mode, gauges provide a snapshot of our system at the moment the metric is taken: So if we're not running much on a system, then start a memory-intensive program (like `stress -m 1`), we would see a drop in the `node_memory_MemFree_bytes` data.

The memory-based data is comparatively straight-forward: To get really usable and understandable data, we often don't even have to do any math, and the math that we do have to perform is simple operations like addition, subtraction, division, and multiplication. That said, `node_memory` also has a number of other available options for many of the more detailed metrics you may need.

## Disk Metrics

Before we begin this lesson, it is suggested you log in to your server and run a stress test against your disk. To do this, run:

```
stress -i 40
```

Disk data is retrieved from `/prod/diskstats` on the server and is specifically related to the performance of our disks. We've seen a touch of I/O information before — specifically, we had an `iowait` mode for our `node_cpu_seconds_total` metric. But what does that actually mean, in greater detail? It's that `iowait` measures the amount of time a CPU is idle due to waiting on the disk itself to complete operations. So this one specific CPU metric is directly related to the metrics we can find regarding our disks.

But Prometheus — as most metrics systems do — tracks more than just I/O wait. Our disk metrics themselves are all fairly basic, providing us with one gauge showing our I/O activity over time, and then providing us with a number of counters tracking both how many completed I/Os have occurred, and then recording the processing time it took to do so.

It should also be noted that Prometheus automatically filters out any loopback devices and partitions. To see what this means, all we have to do is check out `iostat` on our server:

```
iostat
```

Notice how we have five `loop` devices and one device called `xvda`; all of those `loop` devices are filtered through Prometheus. Let's go and view the `node_disk_io_now` metric in our expression editor to see this in action: If you go to the graph, you'll see only a single line. Should we hover over that line, we only have information on one device: `xvda`, the only non-loop disk we saw when running `iostat`.

Similar to with our CPU, the `node_disk_io_time_seconds_total` metrics lets us use `rate` or `irate` with our I/O data to calculate the disk usage for our host:

```
irate(node_disk_io_time_seconds_total[30s])
```

We're also provided with the total amount of time the disk has spent on I/O operations with:

```
node_disk_io_time_seconds_total.
```

If we use this alongside `node_disk_read_time_seconds_total` and `node_disk_write_time_seconds_total`, we can calculate the percent of time our disk is spending on reads and writes, respectively.

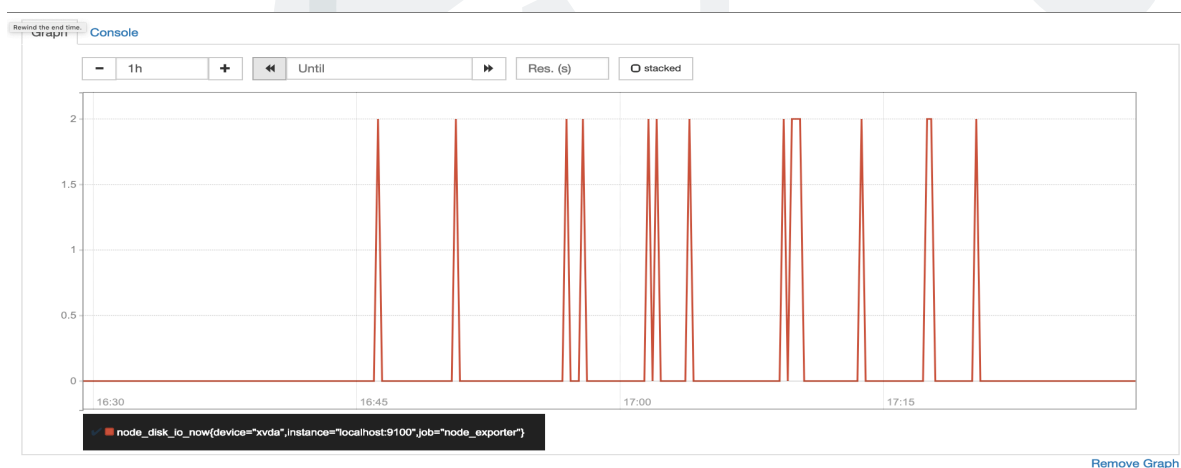
Reads:

```
irate(node_disk_read_time_seconds_total[30s]) / irate(node_disk_io_time_seconds_total[30s])
```

Writes:

```
irate(node_disk_write_time_seconds_total[30s]) / irate(node_disk_io_time_seconds_total[30s])
```

Finally, we do want to review the `node_disk_io_now` metric: This is our gauge. If we go ahead and view it now in our expression editor, we'll be met with a number of spikes caused by our stress test:



## File System Metrics

File system metrics are some of the most straightforward metrics we can deal with in Prometheus and any monitoring solution. That said, because we have multiple file systems mounted, including a container volume and some temporary file systems, it gives us the ability to see some features that we otherwise haven't been able to work with on Prometheus due to our limited setup.

Beginning with the `node_filesystem` prefix, we are provided with seven metrics specific to our file systems. These metrics are restricted to our *mounted* systems only, and many are self-evident, although not without some needed clarifications.

For example:

```
node_filesystem_avail_bytes and node_filesystem_free_bytes
```

May seem like they are calculating the same thing, but in certain Unix systems, an amount of free memory is reserved for the *root* user, so if we wanted to see how much free space we have for our users, we would want to use the `node_filesystem_avail_bytes`, which does not include this *root*-specific space.

Additionally, we want to note that `node_filesystem_files` is specifically related to how many *inodes* are used. Similarly, `node_filesystem_files_free` is how many inodes are free.

Now, let's actually take a look at `node_filesystem_avail_bytes` in our expression editor. Chances are, this is not particularly interesting on your current system, so let's review this graph of six days of metric data from my own demo system:



Notice that we have multiple lines on this chart, just as we did in the CPU lesson. However, what we didn't address is that we can actually manipulate these lines somewhat. For example, if we only wanted to see the information about our `xvda` mount point, we could uncheck the other options:

```
node_filesystem_avail_bytes(device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node_exporter",mountpoint="/run/user/1001")
node_filesystem_avail_bytes(device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node_exporter",mountpoint="/run/lock")
node_filesystem_avail_bytes(device="tmpfs",fstype="tmpfs",instance="localhost:9100",job="node_exporter",mountpoint="/run")
node_filesystem_avail_bytes(device="xfs",fstype="fuse.lxcfs",instance="localhost:9100",job="node_exporter",mountpoint="/var/lib/xcfs")
node_filesystem_avail_bytes(device="/dev/xvda1",fstype="ext4",instance="localhost:9100",job="node_exporter",mountpoint="/")
```

Our file system collector also deviates from our other metrics in that we can add more *labels* than prior metrics. Labels are the part of the metric contained in the curly brackets for each element:

```
{device="/dev/xvda1",fstype="ext4",instance="localhost:9100",\
job="node_exporter",mountpoint="/"}
```

Are the labels for our `xvda` mount, for example. Unlike other metrics, we now have data about the file system type (`fstype`) and mount points (`mountpoint`).

As we saw before, we can use these labels in our expression editor to view section of our data. For example, if we only wanted to view temporary file system data, we could use:

```
node_filesystem_avail_bytes{fstype="tmpfs"}
```

Of course, these options aren't just limited to paring down our file system data. We can use these for any metric!

## Network Metrics

Before we begin this section, we should clarify that the *network* metrics provided by the Node Exporter and full *networking* monitoring are not completely the same thing. Our node exporter gives us a look into our network devices and bytes that are being transmitted and received, but if we want to look into detailed network monitoring, we would need to set up the SNMP exporter. This advanced use-case is out of scope for this course. We will only be discussing network metrics as far as the node exporter takes us.

The node exporter pulls our network device information from `/proc/net/dev`, as well as `/sys/class/net`, as do other monitoring solutions. Within Prometheus, it is prefixed with `node_network`, and our `node_network` elements contain information regarding our addresses (`address`), broadcast address (`broadcast`), operation state (`operstate`), duplex (`duplex`), and interface (`interface`).

With regards to the actual metric provided, many provide a basic insight into the core networking stats you would expect to need as a systems admin or engineer, not a networking engineer. We have simple data like whether a network is up (`node_network_up`), the interface type (`node_network_address_assign_type`), the carrier (`node_network_carrier`), and others. In fact, if we were to list out the contents of `/sys/class/net/` for any one of our interfaces — such as `eth0` — we can see that Prometheus will provide us with data for the majority of these options across a timeseries; all we would have to do is input `node_network_INTERFACEDETAIL` into the expression editor. So if we wanted to know the dormancy status of our network, it would just be `node_network_dormant`.

But with regard to data that we might actually use and manipulate, the metrics that begin with `node_network_transmit` and `node_network_receive` will contain the most of our pertinent data. These metrics specifically contain information about the bytes and packets of data we are accepting on our server or are being sent out.

Specifically, we'll most likely want to be able to calculate the bandwidth of our networks. To do this, we just have to use our `rate` function against either `node_network_receive_bytes_total` or `node_network_transmit_bytes_total` metrics. So if we wanted to see the 30 second average of transmit bytes, we would just run:

```
rate(node_network_transmit_bytes_total[30s])
```

We can also do this same thing with packets:

```
rate(node_network_transmit_packets_total[30s])
```

## Load Metrics

When we discuss “load” in relation to our server, we’re specifically referring to any processes waiting to be served by the CPU. These metrics are taken at 1, 5, and 15 minute intervals, and you’ve probably seen them before: Any time you run the `top` command, these load averages are supplied to us at the top of the screen. Additionally, we can access them by viewing the `/proc/loadavg` file.

On Prometheus itself, we can access our load averages through three metrics: `node_load1`, `node_load5`, and `node_load15`. All of these are simple gauges showing us the point-in-time metrics for each load time.

That said, while these metrics might provide us with some insight to how our servers manage the amount of processes being run, they are mostly useless. We should not be alerting on these at all, and even the raw data won't tell us much: After all, a machine with four CPUs and 16 GB of RAM is going to be able to handle a much higher load than one of our single micro Cloud Playground servers. And since we only have our 1/5/15 metrics, we don't have any "max" per-server baseline we can compare this against.

That said, it's not uncommon to *think* load is more important than it actually is, so we've taken the time to address it. But in your day-to-day monitoring, you should probably leave load for just reviewing processing trends, nothing more.

## Using cAdvisor to Monitor Containers

We've taken a look at quite a few common metrics and how to manipulate them in the past series of lessons. But thus far, we've just been monitoring metrics on the server itself — what about the containers we host on our servers?

For container monitoring, we're going to bring in yet another tool, although this one will be a quick and simple deploy: cAdvisor is a tool released by Google that natively monitors Docker containers and can be used with a number of other container platforms out of the box. We barely even have to do any work to deploy it! All we have to do is run:

```
sudo docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --volume=/dev/disk/:/dev/disk:ro \
  --publish=8000:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

Now, if we take a peak at our list of containers, we can see that we have a second container up and running, called `cadvisor`:

```
docker ps
```

The above command also automatically mapped the cAdvisor container to port 8000 on our host, so we can view it by navigating to the lab server URL and accessing the appropriate port. This is also where Prometheus will check for these container metrics. Let's actually go ahead and add cAdvisor to Prometheus now:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)



```
$ sudo $EDITOR /etc/prometheus/prometheus.yml
```

```
- job_name: 'cadvisor'
  static_configs:
    - targets: ['localhost:8000']
```

Next, restart Prometheus:

```
sudo systemctl restart prometheus
```

Now, when we move to our Prometheus web UI, we can search for the prefix `container` and see a number of options! Another thing you might notice is that many of these metrics are the same or similar to the ones we've already discussed. We have CPU data, although with `cAdvisor` the data is paired down further and can be separated by system and user; file system data that uses the straight-forward `inodes` name instead of `files` to show us how many inodes are used or free; a selection of basic memory information, mostly related to errors and usage; and inbound and outbound networking data.

We can also make sure we're just getting data about the containers we want by using the `name` label. So if we want to see the metrics specific to the memory usage of our application, we can search the expression editor for:

```
container_memory_usage_bytes{name="ft-app"}
```

With the `name` value simply taken from our container name itself that we provided Docker.

## Using the prom-client

Up until this point, we've been dealing with metrics that we, ourselves, did not have to write or add. But when we write custom applications, we want to be able to monitor those, too. And for custom applications, there's not going to be a Node Exporter or `cAdvisor` for us to download. Instead, we're going to have to write these instrumentations ourselves.

This is where Prometheus client libraries come in. Official libraries exist for Go, Java/Scala, Python, and Ruby, although third-party clients exist for a number of other languages, including Bash, C++, Lisp, Elixir, Erlang, Haskell, Lua, .NET, Node.js, Perl, PHP, and Rust. Since our application is written in Node.js, it's safe to guess which one we'll be using.

Before we begin working on our application, however, I do want to note that anything we learn in this section can be adapted for any other language. The concepts behind using a Prometheus client library are the same regardless of language. Although our application is written in Node.js, I personally got familiar with the concepts by looking at Python tutorials, which were better-written at the time. The details of the language might change, but *where* we include our Prometheus counters and tracking in our application won't change.

So, let's go ahead and get started. Move into the `forethought` directory from the `cloud_user`'s home directory:

```
cd forefront
```



If you remember way back to our first real lesson, this is where our application is stored. Normally, while we begin to write our instrumentations, we would be working on a development server. But since we're limited with our test environment, we're going to make our changes straight to this directory itself and testing from here. But before we can do even that, we need to grab the Prometheus client for Node.js:

```
npm install prom-client --save
```

With that finished, we can start by adding it to our application. We'll be working exclusively with the `index.js` file. Go ahead and open this file:

```
$EDITOR index.js
```

We now want to make sure our application has access to the Prometheus client, so we can begin adding metrics. To do this, we need to include the client in our code:

```
const prom = require('prom-client');
```

This should be added after our existing variable `list` (`var app = express();`).

The Prometheus client also offers us the option to collect some default metrics — things like active requests, start time, and memory usage by the application itself. To do this, we just have to add the following to our code:

```
const collectDefaultMetrics = prom.collectDefaultMetrics;  
collectDefaultMetrics({ prefix: 'forethought' });
```

Where `prom` references the constant variable we just defined. The `prefix` setting just defines the prefix for the metric name.

However, we currently have no way to see these default metrics. After all, up until this point we've only seen our metrics after they've been revealed on a `/metrics` endpoint. But our application doesn't have one of these.

Lucky for us, our application already has the ability to create a simple web server. If you look at our initial set of variables, we include Express, a Node.js/JavaScript framework for deploying simple websites. It's how we have the Forethought application up and running despite having neither Apache or Nginx installed (seriously, go look: There's no HTTP server on our host).

If we look further into our code, we can see that we render our website in only a few simple lines, here:

```
app.get("/", function (req, res) {  
  res.render("index", { task: task, complete: complete });  
});
```

We can do this same thing for our `/metrics` endpoint. So we can establish that we want a page called `/metrics` by creating a callback function for any related requests and responses. This is our *route definition*:

```
app.get('/metrics', function (req, res) {  
  });
```

And then within that definition, we can provide the code related to `prom-client` that will provide the response objects that will generate our metrics:

```
app.get('/metrics', function (req, res) {  
  res.set('Content-Type', prom.register.contentType);  
  res.end(prom.register.metrics());  
});
```

Save and exit the file. Now we can go ahead and test our work by running:

```
node index.js
```

This brings up our application on port 8080. To see that our `/metrics` page is available and we've started generating the default metrics, navigate to `YOURLABSERVER:8080/metrics`.

## Counters

In our “Infrastructure Monitoring” section, we frequently used the terms “counters” and “gauges” to discern how the metrics are displaying our data and what that data meant. But now that we're the ones writing these metrics, we want to be sure we really understand what we're having our metrics track — and how.

Counters are generally considered the most commonly used instrumentation method. We saw a counter used within every one of our infrastructure monitoring domains (except load), and that's because when paired with a time series, we can calculate a number of trends and additional metrics based on how often or how little the number is increasing.

And think about how many things within an application we can use a counter for: Connections, any requests made that are specific to the application, errors thrown, and more. In our case, we're going to add a counter to our application that tracks how many total to-do list items have been logged in our application — so every time we click “Add”, the number of this metric would increase by one, allowing us to track how active our application is at any given time.

Let's go ahead and open up our `index.js` file again:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
cd forethought
```

```
$EDITOR index.js
```

When we define a new metric with our Prometheus client, we need to first import the function and provide it with the metric name and the “help” line explaining that metric's purpose. For our Node.js application, this means that every time we add a metric, we'll be following this structure:

```
const METRICNAME = new prom.METRICTYPE
  name: 'NAME_OF_METRIC',
  help: 'HELP INFORMATION'
});
```

To import a counter for our “amount of to-dos” metric, we’d add the following:

```
// Prometheus metric definitions
const todocounter = new prom.Counter({
  name: 'forethought_number_of_todos_total',
  help: 'The number of items added to the to-do list, total'
});
```

Add this near the top of the file, after our variable definitions where we import in the various libraries we use. I’ll be storing all my metrics in this one section, so I added a comment at the top to clarify.

But the Prometheus client isn’t going to inherently know *what* is in our application we’re counting — this is only where we define our counter. To actually have our app start counting up with each new to-do, we need to locate the part of our app that lets us add these to-do tasks:

```
// add a task
app.post("/addtask", function(req, res) {
  var newTask = req.body.newtask;
  task.push(newTask);
  res.redirect("/");
});
```

And call our counter in the middle of the function:

```
// add a task
app.post("/addtask", function(req, res) {
  var newTask = req.body.newtask;
  task.push(newTask);
  res.redirect("/");
  todocounter.inc();
});
```

We use `todocounter` to call the `prom.Counter` function we just created, with `.inc()` signaling that we’re increasing the count. By default, this increases by one, but we can change this by adding the number we want the count to increase in by the parenthesis (i.e., `.inc(2)` would increase by two).

Now let’s see this in action by saving and exiting the file, and running:

```
node index.js
```

Access the application (make sure you're on port 8080 and not the live application!) and add some to-dos to increase the counter, then check the `/metrics` endpoint. Our new task counter has been keeping track of how many to-dos we've added!

## Gauges

The other metric we've been exposed to up until this point has been gauges — these keep track of the state of our system the moment the metric is taken, meaning it isn't a consistent count up like our counters, but a number that can be increased or decreased as the state of our metric changes. The number the gauge tracks can be anything measurable — from how much memory is being used, as we saw before, to how many concurrent connections there are, to how long a request takes to process.

For our demo application, we want to track just how many unfinished to-dos we have: So it keeps track of any tasks added.

We can add a gauge-based metric to our application in a way that is similar to how we added a counter. We first want to define the metric:

```
const todogauge = new prom.Gauge ({
  name: 'forethought_current_todos',
  help: 'Amount of incomplete tasks'
});
```

Then we can call the metric in our code. However, unlike in our previous lesson, we have to make sure we're including both a decrease and an increase:

```
// add a task
app.post("/addtask", function(req, res) {
  var newTask = req.body.newtask;
  task.push(newTask);
  res.redirect("/");
  todocounter.inc();
  todogauge.inc();
});

// remove a task
app.post("/removetask", function(req, res) {
  var completeTask = req.body.check;
  if (typeof completeTask === "string") {
    complete.push(completeTask);
    task.splice(task.indexOf(completeTask), 1);
  }
  else if (typeof completeTask === "object") {
    for (var i = 0; i < completeTask.length; i++) {
      complete.push(completeTask[i]);
      task.splice(task.indexOf(completeTask[i]), 1);
    }
  }
});
```

```
    todogauge.dec();  
  }  
}
```

In this change, the `todogauge.inc()` functions work the same as the `todocounter` function. `todogauge.dec()` just decreases the count instead of increases.

Save and exit, then run `node index.js`. Review the application in your web browser.

But gauges also offer some more advanced utilities, as well. For example, beyond just measuring the amount of tasks we're adding and completing, we can keep track of how long the requests take with the use of the `.startTimer()` function for gauges. However, we're not going to be using a gauge to measure our response time. Instead, we want to use a histogram.

## Histograms and Summaries

Both histograms and summaries are more complex metric types. While all our metrics up until this point have been storing data across a single time series, summaries and histograms work across multiple time series, keeping track of both sample observations — like how long it took to make a request — and the sum of these values, essentially working like a combination of a gauge and a counter.

Summaries, specifically, work by observing and recording an event, such as latency or size, and work in percentiles. When we define a summary, we can define which percentiles we want to use, and the data we take will be placed in the appropriate "bucket" for the percentile.

Histograms work similarly only its buckets aren't restricted to percentages. Histograms allow us to see how a metric happens, proportionally, in buckets.

For example, here's a snapshot of a histogram I have running for a demo version of this course:

```
forethought_requests_bucket{le="0.005"} 0  
forethought_requests_bucket{le="0.01"} 0  
forethought_requests_bucket{le="0.025"} 0  
forethought_requests_bucket{le="0.05"} 0  
forethought_requests_bucket{le="0.1"} 13  
forethought_requests_bucket{le="0.25"} 31826  
forethought_requests_bucket{le="0.5"} 31982  
forethought_requests_bucket{le="1"} 34817  
forethought_requests_bucket{le="2.5"} 34857  
forethought_requests_bucket{le="5"} 34882  
forethought_requests_bucket{le="10"} 34903  
forethought_requests_bucket{le="+Inf"} 34910  
forethought_requests_sum 8397.841003000045  
forethought_requests_count 34910
```

We can see that 13 request took between 0 and .1 milliseconds, 31826 between 0 and .25, and so on. Notice a start with 0 got both of these listed buckets: This is because histograms are cumulative in

Prometheus, so that .25 millisecond bucket includes everything from the .1 bucket and below, as well. Also notice that `_sum` and `_count` metrics are included — this is what we mean by saying two time series, with `_sum` being the sum of all values passed to Prometheus, and `_count` being the amount of calls made for the metric (these are done with `observe`, as we'll see shortly).

But before we can see this in action, let's go ahead and write the code to create our summary and histogram. We're going to be measuring the time it takes for each request made to our application, or the latency.

Before we begin, move into the `forethought` directory and use `npm` to install the `response-time` client — we'll use this to measure our request times for us:

```
cd forefront
```

```
npm install response-time --save
```

Next, open the `index.js` file:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
$EDITOR index.js
```

We can now define our summary and our histogram just as we have in the previous lessons:

```
const tasksumm = new prom.Summary ({
  name: 'forethought_requests_summ',
  help: 'Latency in percentiles',
});
const taskhisto = new prom.Histogram ({
  name: 'forethought_requests_hist',
  help: 'Latency in history form',
});
```

Now comes the part that begins to differ from previous lessons — we're not just simply increasing or decreasing a value here, we're passing in a number that is returned via the `response-time` middleware we just installed. To call our `response-time` function, we'll first create a variable:

```
var responseTime = require('response-time');
```

Then we'll write the function itself, where `time` is the response time in milliseconds:

```
app.use(responseTime(function (req, res, time) {
}));
```

Now we can call our `tasksumm` and `taskhisto` metrics, passing `time` in as their value:

```
app.use(responseTime(function (req, res, time) {
  tasksumm.observe(time);
  taskhisto.observe(time);
})));
```

Save and exit the file. Let's check our response times:

```
node index.js
```

Next, visit the website and perform a few tasks, then view the `/metrics` endpoint. Prometheus has automatically sorted our response times into percentiles in the summary and time-based buffers in the histogram. Right now, these are using the default buckets. But you might notice that we're not using all the buckets we're provided. When it comes to histograms, we'll generally need to do some tweaking regarding bucket sizes. Let's go ahead and do this now.

Use **CTRL+C** to cancel the `node index.js` command, then reopen the `index.js` file. Generally, when we decide which buckets to use, we'll have more than a few minutes' worth of data. Because of this, we're going to base our bucket off the example data from the demo server I've had running for some time. This means we'll be using buckets for .1, .25, .5, 1, 2.5, 5, and 10. To adjust the buckets, we can add the `buckets` parameter to our metric definition:

```
const taskhisto = new prom.Histogram ({
  name: 'forethought_requests_hist',
  help: 'Latency in history form',
  buckets: [0.1, 0.25, 0.5, 1, 2.5, 5, 10]
});
```

Save and exit, then run `node index.js` again. Visit the demo website on port 8080 and perform some tasks to generate traffic.

Now, when we view our `/metrics` endpoint, we can see we have only the buckets we defined.

## Redeploy the Application

Before we continue learning, we do want to make sure we're using the newest version of our application. So it's time for us to do some housecleaning and replace our current Docker container!

Stop the current Docker container for our application:

```
docker stop ft-app
```

Remove the container:

```
docker rm ft-app
```

Remove the image:

```
docker image rm forethought
```

Rebuild the image:

```
docker build -t forethought .
```

Deploy the new container:

```
docker run --name ft-app -p 80:8080 -d forethought
```

Our application is now up and running with our new metrics! Remember to use `docker start ft-app` to restart the application after your Playground server shuts down.

Finally, let's add our application endpoint to Prometheus:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
$ sudo $EDITOR /etc/prometheus/prometheus.yml
```

```
- job_name: 'forethought'
  static_configs:
    - targets: ['localhost:80']
```

Save and exit, then restart Prometheus:

```
sudo systemctl restart prometheus
```

## Rules

Now that we have monitoring set up on all levels of our stack, we want to start taking these metrics and determining when and how we should be alerting on them. We already noted that we want to alert on the *problem* not the symptom, but as we start adding monitoring to our stack, we might often find we don't know what the problems are and everything looks like a potential symptom. So instead, it's best that we begin to craft our alerting rules based on what affects the end user.

Who is the end user in this case? Our application's users, of course. But while making sure our users can access our application is paramount, we also have to make sure our internal end users are taken care of. After all, what good is all the work we've put in our monitoring system if all our instances of cAdvisor are down?

With Prometheus itself, when we set up alerting, we work in two parts: We define the alerting rules, which are native to Prometheus and can be included regardless of alerting tool; then, we set up the Alertmanager itself to group, manage, and then pass on these alerts to their appropriate endpoint.

So what is a good, baseline metric to alert on? How about checking for how many of our application instances are up? If a large portion of our application containers are down, there's a good chance our end users will notice, which means we need to know about it right away. So how are we going to check for this?

Let's check out our expression editor to start. To see which of our targets are up and running, we can just



input:

```
up
```

But we don't want to alert on just anything that's a target. We also need to think about what will happen when we have multiple instances doing the same job — in real life we're going to have more than one application container, after all. So instead of relying on a simple "Is it up?" check, we want to determine the percentage of down containers that are concerning to us — let's say 25%, although we may find we have to tweak this number as we go on. In this case, we want to take the average:

```
avg (up)
```

But now we have a problem! That just averaged *everything*, not just our forethought container. We can, however, break this out so that we're getting an average per-job by using the `without` clause, which lets us define which parts of our query we want to keep. In this case, we don't want everything to average together by instance, so we can update our query with:

```
avg without (instance) (up)
```

We're still not done, however. Since we just want the average of our forethought jobs, we need to specify this:

```
avg without (instance) (up{job="forethought"})
```

Now, since our forethought job is up and running — and there is only one of them — our average number is 1. We want to alert in the instance this number drops below .75, which means we only have 75% of our forethought containers working:

```
avg without (instance) (up{job="forethought"}) < .75
```

Of course, to actually set up our alert, we need to move from our expression editor to our command line, and open up our Prometheus configuration file:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/prometheus/prometheus.yml
```

From here, locate the `rule_files` section of the configuration. This is where we're going to define the file that will contain our alerting rules. There can be one file or multiple, depending on our setup. One suggestion would be to have a rules file per team that needs alerting — networking would be able to define their own requirements, as would front-end, systems, etc. Since there's just us, however, we're going to be using a single rules file called `rules.yml`:

```
rule_files:
  - "rules.yml"
```

Save and exit the file.

Next, we want to create the `rules.yml` file itself:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/prometheus/rules.yml
```

Rules are written in YAML and are *always* organized in groups, with a group name that cannot be reused across the rules namespace. Rules in a group are always run together. Let's go ahead and name our group `uptime`, since we're using it to track the uptime of anything that will affect our end users:

```
groups:
  - name: uptime
```

Next, we want to add the rules themselves. But before we create an *alerting* rule, let's just create a *recording* rule. A recording rule pre-calculates any statistics we want to keep frequent track of. Essentially, it runs a PromQL query and records the results for us at the same interval we defined in the configuration file. So if we wanted to record that avg without (instance) (up{job="forethought"}) metric so we always have a sense of what percent of our application is down, we would create the following recording rule:

```
groups:
  - name: uptime
    rules:
      - record: job:uptime:average:ft
        expr: avg without (instance) (up{job="forethought"})
```

With `record` being a name we provide to reference the recording, and `expr` being the PromQL expression we want to keep track of. Notice that format of the name, as well:

Let's now save and exit our rules file and restart Prometheus:

```
sudo systemctl restart prometheus
```

Return to the Prometheus web UI. Under **Status**, click on the **Rules** link. We now have our `job:uptime:average:ft` expression recorded. When we click on the rule itself, it takes us to the expression editor for our rule: Notice how it doesn't have our expression in the editor, but the name of the rule itself. Once we add a rule name, we can call the expression by the name in the editor, as though it were written in PromQL.

But how do we turn this recording rule into an alert? First, we need to reopen our rules file:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/prometheus/rules.yml
```

We now want to define an *alerting* rule, which will work always exactly like a recording rule, but we use the parameter `alert` instead of `record`:

```
groups:
- name: uptime
  rules:
  - record: job:uptime:average:ft
    expr: avg without (instance) (up{job="forethought"})
  - alert: ForethoughtApplicationDown
    expr: job:uptime:average:ft < .75
```

Notice how we provide a more traditional name for the alert value — something descriptive enough that we know what's going on, but still short. Also notice how, for the `expr`, we call our metric using the recording rule name — this is not necessary, but does let us know this is a recorded metric we're alerting on.

Save and exit the file, then restart Prometheus:

```
sudo systemctl restart prometheus
```

Next, refresh the **Rules** page in our web UI. We can see we have a second rule added to the list, this time for our "less than 75%" metric. This gives us the basis for our alerting rules, but we can take it a step farther.

## For

We now have a basic alert set up for when our servers have less than 75% uptime recorded. But when it comes to setting up alerts, we have far more options than what we addressed in the last section, and creating robust, *useful* alerts often means doing more than just supplying an alert name and expression to check against. Instead, we want to make sure our alerts are set up so whoever is answering these alerts can best succeed in troubleshooting the issue.

Let's now reopen our `rules.yml` file and flesh out our rules:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/prometheus/rules.yml
```

As it stands, our alert is set to fire whenever 25% of our servers are down. However, there might be instances where this is expected. Say, for example, we're deploying a new version of the application and are pushing updates to a percentage of our servers at a time based on time zones — we very well might be updating 25% of our fleet, as expected, but still end up receiving alerts as our targets are restarted. To prevent instances like this, we can make sure our alert only fires when the action is consistent. So if Prometheus runs our alerting rule once and it fails, it won't alert, but if it consistently failed across a five minute period, it will. To add this, we use the `for` condition:

```
groups:
- name: uptime
  rules:
  - record: job:uptime:average:ft
```

```
    expr: avg without (instance) (up{job="forethought"})
- alert: ForethoughtApplicationDown
  expr: job:uptime:average:ft < .75
  for: 5m
```

When an alert fails but does not yet trigger an action, it is considered a *pending* alert. This means it has not yet been sent to Alertmanager and we will not be notified. It will also show up on the **Alerts** page on our web UI.

Generally, you should always have a `for` time set. Five minutes is a great place to start, but you can go longer depending on severity and frequency.

We also want to be able to provide whoever is answering our alerts with as much relevant information as we can. This can be provided using annotations, which let us set a number of key-value pairs that can be filled with whatever additional information we need, either as plain descriptions or using Go's templating system.

## Annotations

We also want to be able to provide whoever is answering our alerts with as much relevant information as we can. This can be provided using annotations, which let us set a number of key-value pairs that can be filled with whatever additional information we need, either as plain descriptions or using Go's templating system. For example, if we want to include what percent of our application's instances are up, we can use:

```
groups:
- name: uptime
  rules:
- record: job:uptime:average:ft
  expr: avg without (instance) (up{job="forethought"})
- alert: ForethoughtApplicationDown
  expr: job:uptime:average:ft < .75
  for: 5m
  annotations:
    overview: '{{printf "%.2f" $value}}% instances are up for {{ $labels.job }}'
```

Notice the two sections of the value in the curly brackets — this indicates that we're using template expressions. With Prometheus, `$value` and `$label` are set variables. `$value` indicates the value of the expression run in the alert, while `$label` can be used to call any label — just add a period and then the label name, as shown for the `job` label.

`printf` is a Go function that will print our expression based on the `"%.2f"` parameters we provided. The `"%.2f"` ensure only the two numbers after the point in our evaluated expression are printed — effectively turning our decimal point into a percentage.

We can also provide more than one annotation. Additional information to include within alerts are links to any documentation on the issue, any debugging information, or links to a relevant Grafana dashboard.

## Labels

Beyond preparing our alerts for the humans who will be responding to them, we also want to make sure they're set up so we can best use them with Alertmanager itself — remember that up until now, all of the rules we've been working with are native to Prometheus, not part of the Alertmanager setup. However, Prometheus still has options for our alerting that will help us when it comes time to bring Alertmanager into the equation.

Alert labels work similarly to annotations — with the exception that while annotations are information and meant more for our engineers and admins, labels work closer with the system itself — specifically, Alertmanager.

Once our Alert makes it to the Alertmanager, we do not want to deal with it individually. Instead, we want to be able to deal with alerts by severity, as well as most likely by team. This will allow us to route our alerts via these labels instead of individually. For our example alert, we want to set the severity to page and the team to devops:

```
groups:
- name: uptime
  rules:
    - record: job:uptime:average:ft
      expr: avg without (instance) (up{job="forethought"})
    - alert: ForethoughtApplicationDown
      expr: job:uptime:average:ft < .75
      for: 5m
      labels:
        severity: page
        team: devops
      annotations:
        overview: '{{printf "%.2f" $value}}% instances are up for {{ $labels.job }}'
```

Save and exit the file. We can now restart Prometheus:

```
sudo systemctl restart prometheus
```

Let's now test our alert by taking down our ft-app container:

```
docker stop ft-app
```

Move to the web UI. To see the state of our alerting, we can click **Alerts**. Right now we have one active alert for ForethoughtApplicationDown, and its state is pending — this goes back to our for setting. Once five minutes are up, this will switch to firing, sending the alert on to Alertmanager.

## Preparing our Receiver

Before we can actually set up the Alertmanager to pass on alerts, we want to set up a place where we can receive them. For this, we're going to be setting up a Slack account — if you already have access to

a workspace you can add applications and test on, just use that and skip this section. Otherwise, we're going to walk through setting up a workspace, adding an app, and configuring it to use webhooks.

First, we want to go to [slack.com](https://slack.com) and click **Sign in**. Do not sign in yet! Instead, click **Create a new workspace** and follow the instructions on the screen — you will need access to your email, but other than this, the process should be quick. When asked for any projects you are working on, provide the channel name of Prometheus.

Once you're placed in your Slack chat, click on the name of the workspace and move to **Administration**, then **Manage apps**.

From here, select **Build** from the upper-right. Press **Start Building**, then **Create New App**. Give your application a name, then select the workspace you just created. Click **Create App** when done.

Next, we need to select **Incoming Webhooks** from the menu. Turn these webhooks **On**, then click **Add New Webhook to Workspace**. Set the channel you wish to post to, then **Authorize** the webhook. We will now be provided with a webhook URL. This is all we need to proceed.

## Using Alertmanager

Now that we have our alerts firing, we need to make sure Alertmanager sends them where they need to go, without overwhelming the ticketing system or on-call with a bunch of ungrouped alerts. Here is where we'll really see the labels we added in our last lesson shine, as we use them to help route, silence, and otherwise manage our alerts.

All of our Alertmanager configurations are stored in a single file, `alertmanager.yml`, which we have placed in our `/etc/alertmanager` directory. Let's go ahead and open this file:

**Note:** Replace `$EDITOR` with the text editor you prefer to use (`vi`, `vim`, `nano`, etc.)

```
sudo $EDITOR /etc/alertmanager/alertmanager.yml
```

As we can see, this file is already broken up into four main parts: The `global` configuration, `routes`, `receivers`, and `inhibition rules`.

Our `global` config exists to store any global parameters we need to provide. Generally, this will be any SMTP information if we're using email to receive any alerts. Since we're not set up to use SMTP on our server, we can leave this with only our `resolve_time` — which is how long an unanswered alert will exist before it is declared resolved. So if we were to bring our container back up, our alert would automatically clear after five minutes.

Next, we have our `route` definitions. Right now, we're only set up for one, by default, but we'll generally need more than this. Let's actually go ahead and rewrite this section so it will better reflect how we'll need to manage alerts.

Chances are, if we're setting up things by severity, that severity will dictate *where* the alert will be sent. Our current alert will be set to a severity of `page`, which tells us we want it sent to some kind of paging utility, such as PagerDuty or OpsGenie. But since we aren't set up for those, we're instead going to route

our pages to our Slack chat.

Let's go ahead and expand the existing `route` parameters, and start by defining a `match` — this will tell Alertmanager we want our route to first match the label we provide for it to work with the alert:

```
route:
  receiver: 'slack'
  group_by: ['alertname']
  group_wait: 10s
  group_interval: 10s
  repeat_interval: 1m
  routes:
    - match:
        severity: page
        group_by: ['team']
        receiver: 'slack'
```

Notice how we provide some default values in case we add alerts that fall outside of our expected labels. Should we receive an alert *not* of the page severity, it will use initially-defined receiver. On the other hand, if our severity *is* set to page, the alert will be grouped by the team, then sent to our slack receiver. `repeat_interval` is how long Alertmanager will wait before resending an alert.

We can take this a step further, too. If we wanted to route by team, we can add our `routes` parameter underneath the current one we're using:

```
route:
  receiver: 'slack'
  group_by: ['alertname']
  repeat_interval: 1m
  routes:
    - match:
        severity: page
        group_by: ['team']
        receiver: 'slack'
        routes:
          - match:
              team: devops
              receiver: 'slack'
```

Of course, this would better serve us if we actually had more than one receiver set up, but for now we'll just set both to slack.

Next, we need to set our receivers. As before, we'll be better suited by removing our current `receivers` section and starting fresh. We first want to make sure the `name` of our receiver is the same that we use when defining any receivers above:



```
receivers:  
- name: 'slack'
```

From there, our configuration will depend on what we are configuring it for. Since we're using Slack, we'd use the `slack_configs` option to set our configurations. Here is where we'll need our API URL from the webhooks page:

```
- name: 'slack'  
  slack_configs:  
    - channel: "#prometheus"  
      api_url: WEBHOOK
```

We can also pull in any of our annotations with the `.CommonAnnotations` option, allowing us to provide more information to our messages:

```
receivers:  
- name: 'slack'  
  slack_configs:  
    - channel: "#prometheus"  
      api_url: https://hooks.slack.com/services/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  
      text: "Overview: {{ .CommonAnnotations.overview }}"
```

Should we need to add more receivers, we would just follow this format, adding more underneath the `receivers` key.

Finally, we can set our inhibitions — these are what automatically suppress alerts based on certain criteria. For example, if we're receiving an alert for a page-severity task for the DevOps team, we might want to suppress any ticket-level alerts that would be associated.

To do this, we would update the existing inhibition rules so that the `source_match` will match our priority severity with the `target_match` referencing what we want to suppress:

```
inhibit_rules:  
- source_match:  
    severity: 'page'  
  target_match:  
    severity: 'ticket'
```

Finally, we'd set the `equal` match so that it would check to see if it's a team match:

```
inhibit_rules:  
- source_match:  
    severity: 'page'  
  target_match:  
    severity: 'ticket'  
  equal: ['team']
```



And now, with our Alertmanager configured, we can save and exit, then restart the service:

```
sudo systemctl restart alertmanager
```

Let's now navigate to port 9093 on our server to check it out. We can see we have an alert that's been fired and noticed by Alertmanager. But are our routes working?

Well, a quick check of our Slack chat will show us that our alerts are up and firing as intended! Our Alertmanager is configured!

## Silencing Alerts

While most of the hard work of Alertmanager is done through the configuration file we just spent some time updating, we can also do some management through the Alertmanager web UI located at port 9093 of our server. Specifically, the web UI is useful when we need to either manually silence alerts, or schedule silences for planned outages and updates.

To silence a specific alert, we can simply click **Silence** next to the alert. We can set a time for how long we want the silence to occur — remember to give yourself some time to troubleshoot. We will leave this at two hours. From there, we're given a list of "matchers" that let us define the traits we want the silenced alert to have. If we wished to generalize our alert further, we can remove some of these options. We also have the option to use regular expressions. Let's go ahead and remove all the matchers, except "job" — silencing all alerts for the `forethought` endpoint.

We don't want to let people silence alerts without reason, however, and we're required to include both our name and some kind of comment regarding the silence before we can create the alert. Go ahead and input your name. I used "Container stopped; silencing as container is restarted." for the comment.

Click **Preview Alerts** to see a list of which alerts will be silenced. If you're happy with the results, click **Create**.

Since we have the alert silenced, let's go ahead and "troubleshoot and fix the problem" by restarting our application container:

```
docker start ft-app
```

Of course, we don't have to limit ourselves to building silences off of existing alerts. We can always click **New Silence** to create one from scratch.

## Adding a Dashboard

When it comes to setting up visualizations that will remain up and running persistently on our Grafana setup, it might seem overwhelming to consider which of our countless metrics we need to rely on. One way to mitigate this problem is using one of the pre-built dashboards shared to the Grafana community. These dashboards are pre-created setups based on common tools and configurations we frequently see when monitoring various infrastructures, containers, and applications. As much as we want to believe our platform is completely unique — and it, technically, most likely is — more are similar enough that pulling in some prebuilt dashboards is a practical way to start.

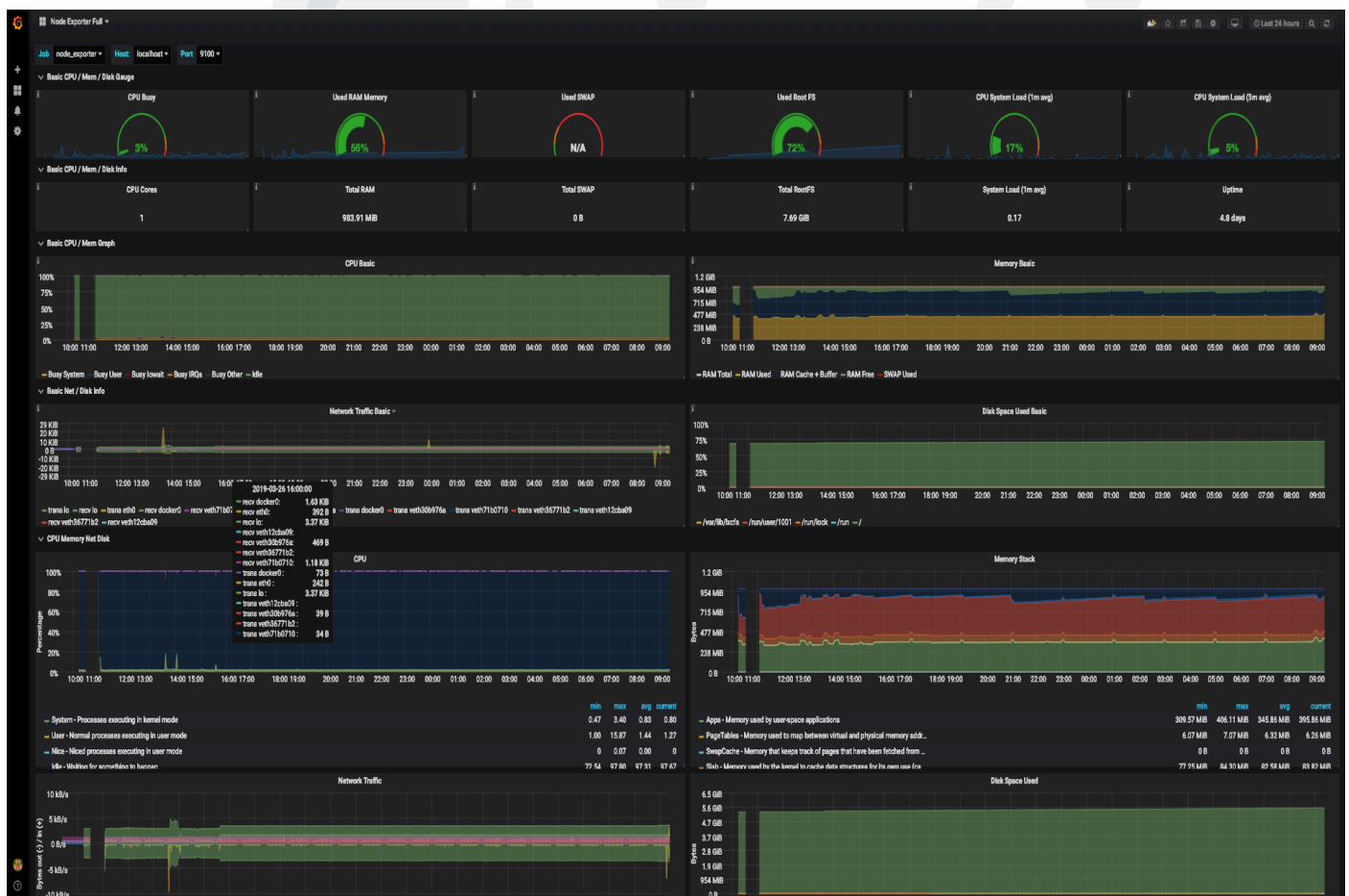
Let's start by tracking down a dashboard for us to install. Go to the [Grafana dashboard page](#). We're given the option to filter this by data source, panel type, category, collector, and by key words.

We want to search for "Node Exporter Full," by idealista. Since these dashboards don't include any comments, reviews, or ratings to judge the quality, it's best to look for dashboards that have a lot of downloads, and have been updated within the past few months. Dashboards aren't something that need to be updated frequently, but a two-year-old dashboard probably isn't going to work as effectively as one that's been updated to the latest release of Grafana.

Once you find the appropriate dashboard, click on it, then select **Copy IP to Clipboard**. Now, we can navigate to our *own* Grafana instance, at port 3000 on our server. Remember to log in as admin, using the password you set at the start of this course.

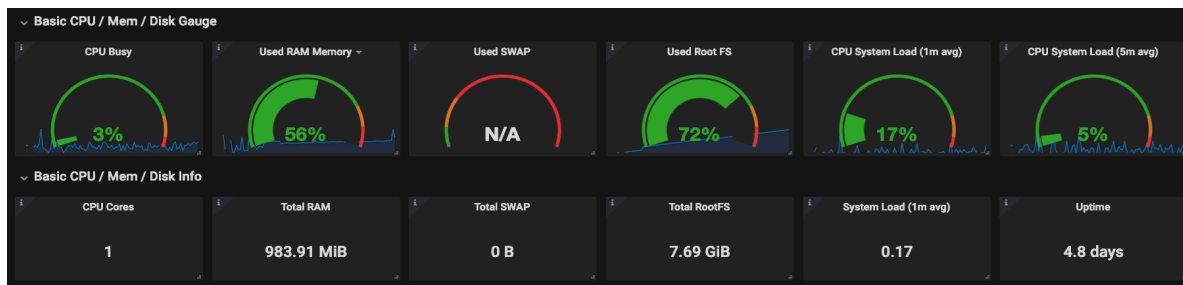
From here, click on the plus sign on the left menu, then click **Import**. Copy the ID into the provided space; the page will automatically readjust with a new form to fill out. Leave the **Name** value as-is, but create a **New Folder** under **Folder**. Give the folder the name *Prometheus*. Leave the **Unique identifier** as-is, as well, then select *Prometheus* under **localhost** to set the data source. Click **Import**.

We're now presented with our dashboard, and as we can see, it's packed with information:



Let's break some of this down to get a better sense of both what is going on, and how Grafana displays data.

## Singlestat Panels



A “singlestat” panel provides us with a summary of a single series or metric. It’s a single, digestible number that provides us with an overview of a statistic. These work especially well with any gauges, although in this dashboard we see them used predominantly to calculate CPU, memory, and disk percentages, as well as to keep track of basic information like CPU cores and the total size of our memory and disk.

To get a better view of how this works, let’s go ahead and click on the title for **Used RAM Memory**, then click **Edit**. We’re taken immediately to the **Metrics** page for defining the metric of the stat.

There are actually two metrics here: The top metric is hidden and thus unused (see the eye icon to the right). The second metric is what’s calculating that percentage in the gauge. Broken down, it’s calculating this expression:

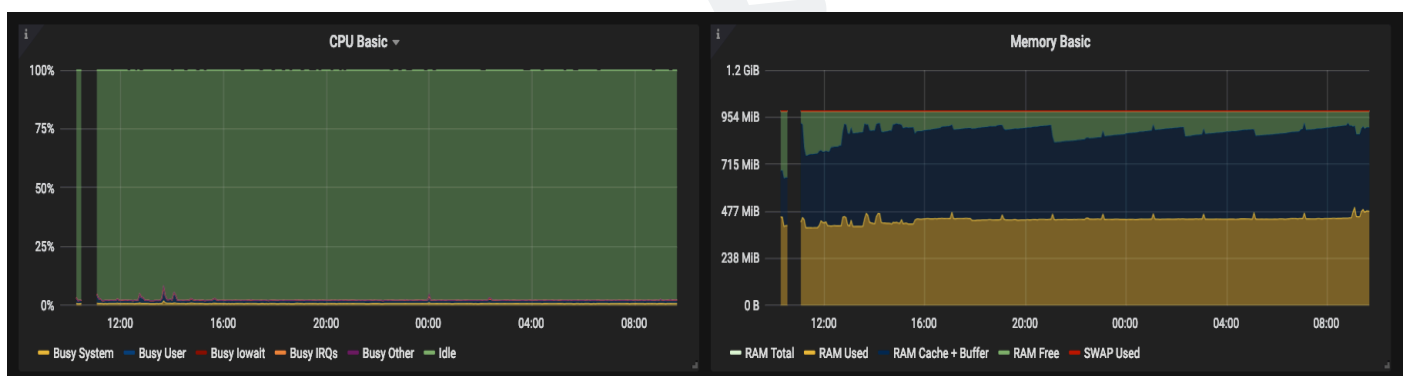
```
100 - ((node_memory_MemAvailable_bytes * 100) / node_memory_MemTotal_bytes)
```

Which gives us our percentage of RAM used. The labels included in the Grafana expression itself are pulled from the **Host** and **Job** drop-downs at the top of the page.

Beyond just adding metrics, we also have additional tabs for our singlestat gauge, where we can provide the title and description (**General**), set the threshold amounts colors (**Options**), and define any value mappers and time ranges (which are not currently being used).

Move back to the main dashboard now.

## Graph Panels



Graphs are the “main panels” of Grafana — these are the detailed panels that include a look at our data over time. We can make some adjustment to these straight from the dashboard — if we click on one of

the color keys, we can change the axis and colors, while if we click on the name of the key, we can toggle that key on or off.

Let's go ahead and click on the title of the **CPU Basic** graph, and select **Edit**. Once more we're taken to the **Metrics** page, where we're met with familiar metrics — we used `rate` against our `node_cpu_seconds_total` metric in one of our very first lessons, and we can see that here we have that metric set up for every single CPU mode label. The only notable difference is that this graph aims to include the data for the sum all *all* instances (`sum by (instance)`) even though we only have a single instance to go off of.

Like with our singlestat gauge, we can also tweak the different parts of this graph in the different tabs provided. Most of these are self-explanatory: **Axis** lets us set up our axis, **Legend** lets us change up our legend, if needed, **Alert** lets us set up an alert. Oftentimes we won't have to make any adjustments here, but the option is there.

## Altering the Dashboard

Just because we imported a dashboard doesn't mean we're stuck with it as-is. We've already viewed how to edit our graphs, but what happens when we need to remove something or make any lasting changes?

Scroll down and open the **System Detail** list. At the bottom of the section, we have a graph measuring hardware temperatures, which is not data we have, so let's go ahead and click on that graph title, then select **Remove**.

Now we want to save our updated dashboard. We can do this by clicking the save icon at the top of the screen. We're asked to include a small comment about the change, for the sake of tracking. Go ahead and do this, then save.

## Suggested Dashboards

When it comes to our Grafana setup, we probably aren't just going to be pulling in Prometheus data, however. As such, I do want to mention one other way to figure out which premade dashboards to check out. If we go to the cog icon on the left menu and click on **Data Source**, then select a data source, we can view the **Dashboards** tab to see any official Grafana-provided dashboards available for use.

Of course, while these premade dashboards are a great place to start, we also want to be able to create our own graphs, heatmaps, gauges, and more, which we'll be covering next.

## Building a Panel

Once we get some of the simpler parts of our Grafana setup configured, we'll want to start looking into adding metrics we won't find any pre-created dashboards for, such as those metrics related specifically to our application. While using a default Node Exporter dashboard saves us a lot of time and works well for monitoring our infrastructure metrics, we don't want to leave our applications in the dark.

Let's switch to the **Forethought** dashboard we created way back at the beginning of the course. We're going to start populating this dashboard with our application metrics, specifically, starting by generating

a heatmap to see the distribution of our response times.

Click on the **Add panel** button on the top menu. Select **Heatmap** from the list. A plain panel is added. Click on the title and select **Edit** to begin building out our new visualization.

Instead of starting on the **Metrics** tab, we're going to begin with **General**. This is just where we set the basic information for our panel, which I'm going to name "Response Time Distribution".

Next, we want to switch to **Metrics**. Let's just start by calling our `forethought_requests_hist_bucket` metric, and setting the **Legend format** to `{{le}}`.

Our graph is automatically populated, but it's not quite what we're looking for. Move to the **Axes** tab. Switch the **Data format** to *Time series buckets*. The Y-axis is switched so it uses our buckets! We're getting closer already.

We still want to better refine our graph, however. While we *can* tell what our most-used buckets are with this heatmap, we can get a better idea of the average response rates over time by using `rate` alongside our expression. We're also going to wrap `sum` around this, so that all instances of this metric are combined — something needed if we want to provide a view of all our application instances, since we'd most commonly have more than one:

```
sum(rate(forethought_requests_hist_bucket[30s]))
```

Unfortunately, our graph becomes unusable after this. Because our expression has gotten more complicated, we need to let Grafana know we're using our `le` buckets:

```
sum(rate(forethought_requests_hist_bucket[30s])) by (le)
```

The resulting graph looks a lot like the original one, but now it shows us not just how many responses fell in each bucket, but also how often these responses occurred.

We can also further tweak the heatmap by using the **Display** tab. Here we can choose to define a legend or set buckets, neither of which we need to do. Instead, take some time to explore the **Colors** section. One option, in particular, that you might want to view is the **Mode** — I find setting it to *opacity* makes it much easier to see, especially if you have any colorblind users!

Return to the Forethought dashboard, and **Save** our new panel. Let's click on the **New panel** icon again. We've already discussed graphs, singlestats, and heatmaps, but what about our other options?

**Tables** are just that — a table of content with rows and columns. We don't have any good data for tables, and you'll probably never need to use them with Prometheus; however, they can be useful for other data sources.

**Text** is pure markdown content. We can write whatever we want here to display on our dashboard.

An **Alert List** lets us include a list of alerts.

A **Dashboard List** works as a list of dashboards. Options are limited here — you can choose to have starred, most recent, and a set of searched dashboards, but no way you can individually add dashboards or sort by folders.

A **row** let's us break down our dashboard into sections — we saw this with our Node Exporter dashboard.

Finally, the **Plugin List** is just a list of any used plugins. We are currently not using any.

The majority of the time, however, you'll find you're using the singlestat, graph, and heatmap utilities.

## Congratulations!

You did it! Congratulations, you've taken the time to learn how to use monitoring concepts at all levels of your platform. You're now prepared to not just set up a monitoring stack, but use metrics, write metrics into code, and manipulate those metrics across various tools, such as Grafana, to keep track of just what is happening “under the hood” of your system.

Not ready to take a break yet? That's okay! We have plenty of courses that will sate that need to learn:

- [YAML Essentials](#)
- [Monitoring Kubernetes with Prometheus](#)
- [Elastic Stack Essentials](#)
- [Elasticsearch Deep Dive](#)