

# Monitoring Spring Boot Apps with Micrometer, Prometheus, and Grafana

By Dhananjay Singh • October 22, 2019 •

0 Comments (/monitoring-spring-boot-apps-with-micrometer-prometheus-and-grafana)

## Introduction

Monitoring an application's health and metrics helps us manage it better, notice unoptimized behavior and get closer to its performance. This especially holds true when we're developing a system with many microservices, where monitoring each service can prove to be crucial when it comes to maintaining our system.

Based on this information, we can draw conclusions and decide which microservice needs to scale if further performance improvements can't be achieved with the current setup.

In this article, we'll cover how to monitor Spring Boot web applications. We will be using three projects to achieve this:

- Micrometer (<https://micrometer.io/>): Exposes the metrics from our application
- Prometheus (<https://prometheus.io/>): Stores our metric data
- Grafana (<https://grafana.com/>): Visualizes our data in graphs

This might look like a lot, especially compared to just using the Spring Boot Actuator (<https://spring.io/guides/gs/actuator-service/>) project, but it's very easy to implement all of them with just a few configurations.

To make things even easier, we'll be using Docker (<https://www.docker.com/>) to run Prometheus and Grafana since they both have official Docker images. If you're not familiar with Docker, you can check out our article [Docker: A High-Level Introduction \(/docker-a-high-level-introduction/\)](/docker-a-high-level-introduction/).

Please note that these metrics will give you aggregated information over an interval of time. If you want to check information about an individual request at a particular time and what happened to it, then this might not be the solution for you.

In that case, you probably need a distributed tracing system which we have covered in detail in [Distributed Tracing with Sleuth \(/spring-cloud-distributed-tracing-with-sleuth\)](/spring-cloud-distributed-tracing-with-sleuth/).

## Spring Boot Actuator

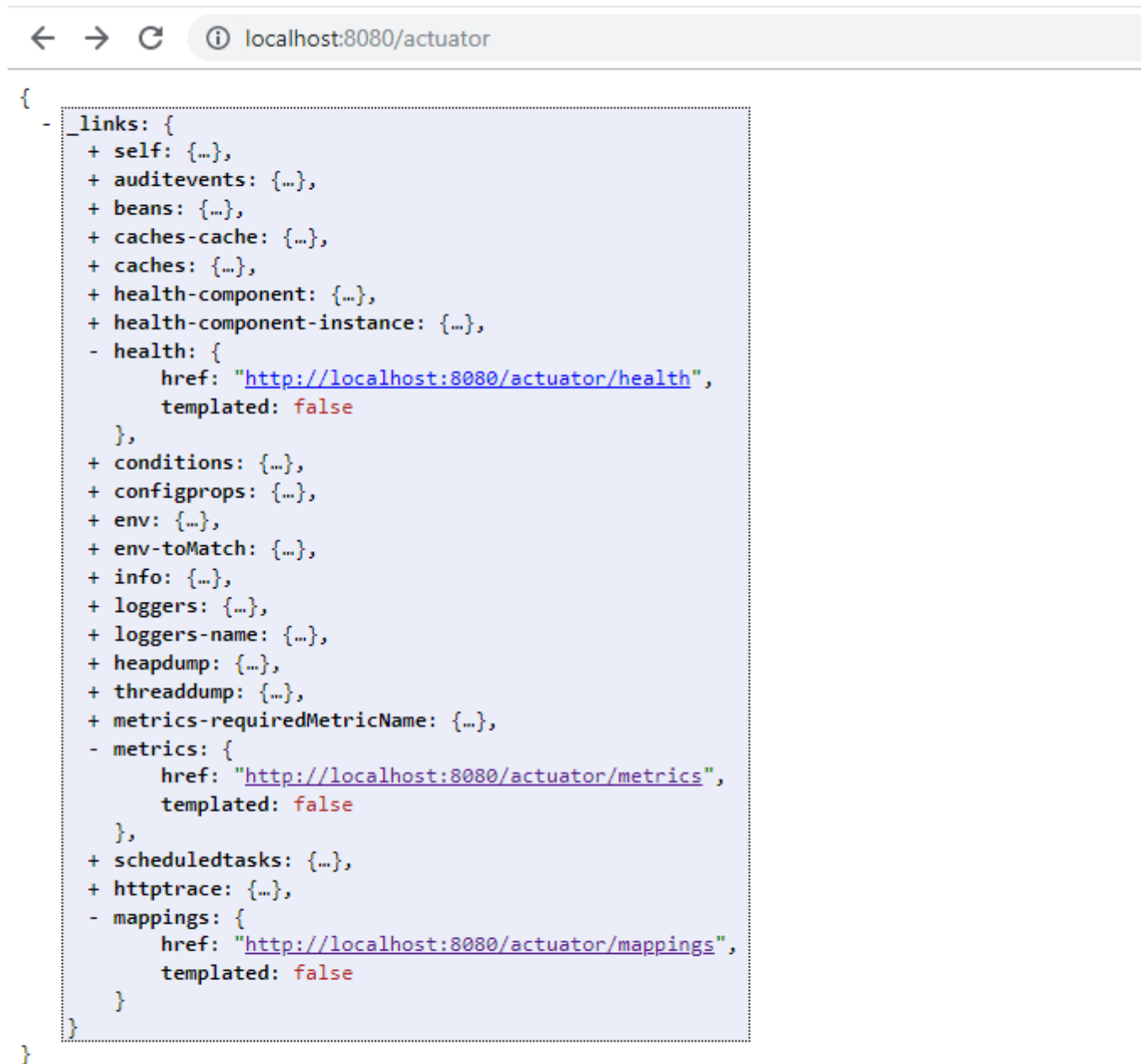
We'll start off with a simple REST service using Spring Initializr (<https://start.spring.io/>) that contains a single endpoint of `/hello` and running on the default port of `8080`.

Besides, this application also has the `spring-boot-starter-actuator` dependency, which provides production-ready endpoints (<https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html>) that you can use for your application. These endpoints fall under a common prefix of `/actuator` and are, by default, protected.

Expose them individually, or all at once, by adding the following properties in `application.properties`:


```
management.endpoints.web.exposure.include=*
```

To check, let's navigate our browser to `http://localhost:8080/actuator`:



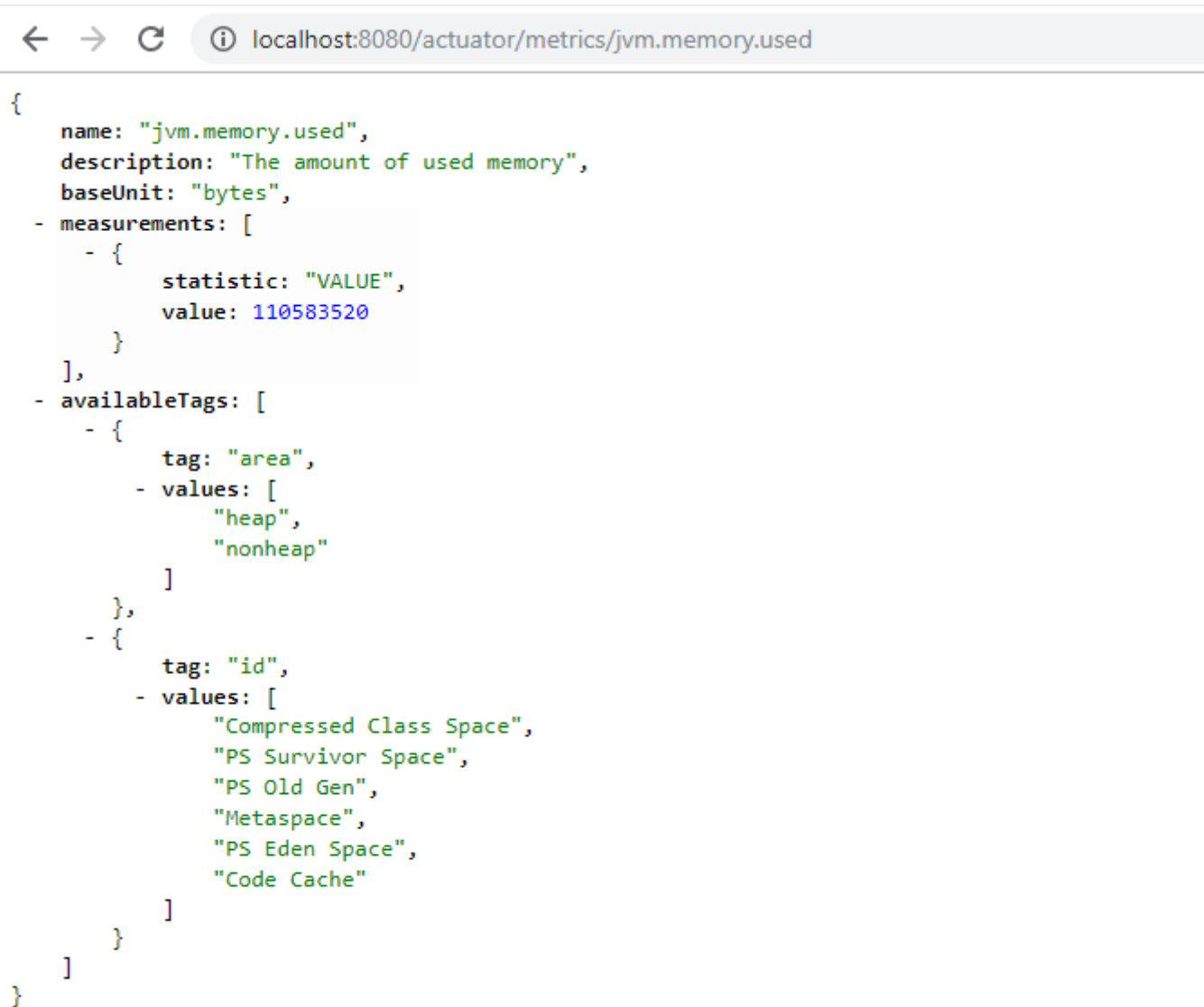
```
{
  - _links: {
    + self: {...},
    + auditevents: {...},
    + beans: {...},
    + caches-cache: {...},
    + caches: {...},
    + health-component: {...},
    + health-component-instance: {...},
    - health: {
      href: "http://localhost:8080/actuator/health",
      templated: false
    },
    + conditions: {...},
    + configprops: {...},
    + env: {...},
    + env-toMatch: {...},
    + info: {...},
    + loggers: {...},
    + loggers-name: {...},
    + heapdump: {...},
    + threaddump: {...},
    + metrics-requiredMetricName: {...},
    - metrics: {
      href: "http://localhost:8080/actuator/metrics",
      templated: false
    },
    + scheduledtasks: {...},
    + httptrace: {...},
    - mappings: {
      href: "http://localhost:8080/actuator/mappings",
      templated: false
    }
  }
}
```

You can see all the endpoints that Actuator exposes such as `/health`, `/metrics`, `/mappings`, etc. Let's open up the `/metrics` endpoint of the Actuator by navigating our browser to `http://localhost:8080/actuator/metrics`:

 localhost:8080/actuator/metrics

```
{
  - names: [
    "jvm.memory.max",
    "jvm.threads.states",
    "jvm.gc.memory.promoted",
    "http.server.requests",
    "jvm.memory.used",
    "jvm.gc.max.data.size",
    "jvm.gc.pause",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "tomcat.global.sent",
    "jvm.buffer.memory.used",
    "tomcat.sessions.created",
    "jvm.threads.daemon",
    "system.cpu.usage",
    "jvm.gc.memory.allocated",
    "tomcat.global.request.max",
    "tomcat.global.request",
    "tomcat.sessions.expired",
    "jvm.threads.live",
    "jvm.threads.peak",
    "tomcat.global.received",
    "process.uptime",
    "tomcat.sessions.rejected",
    "process.cpu.usage",
    "tomcat.threads.config.max",
    "jvm.classes.loaded",
    "jvm.classes.unloaded",
    "tomcat.global.error",
    "tomcat.sessions.active.current",
    "tomcat.sessions.alive.max",
    "jvm.gc.live.data.size",
    "tomcat.threads.current",
    "jvm.buffer.count",
    "jvm.buffer.total.capacity",
    "tomcat.sessions.active.max",
    "tomcat.threads.busy",
    "process.start.time"
  ]
}
```

As you can see, there's a bunch of information about our application here, such as information about threads, Tomcat sessions, classes, the buffer, etc. Let's go deeper and retrieve information about the JVM memory used:



```
{
  name: "jvm.memory.used",
  description: "The amount of used memory",
  baseUnit: "bytes",
  - measurements: [
    - {
      statistic: "VALUE",
      value: 110583520
    }
  ],
  - availableTags: [
    - {
      tag: "area",
      - values: [
        "heap",
        "nonheap"
      ]
    },
    - {
      tag: "id",
      - values: [
        "Compressed Class Space",
        "PS Survivor Space",
        "PS Old Gen",
        "Metaspace",
        "PS Eden Space",
        "Code Cache"
      ]
    }
  ]
}
```

Now, using the Spring Boot Actuator like this does yield a lot of information about your application, but it's not very user-friendly. It can be integrated with Spring Boot Admin (<https://github.com/codecentric/spring-boot-admin>) for visualization, but it has its limitations and is less popular.

Tools like Prometheus, Netflix Atlas (<https://github.com/Netflix/atlas>), and Grafana are more commonly used for the monitoring and visualization and are language/framework-independent.

Each of these tools has its own set of data formats and converting the `/metrics` data for each one would be a pain. To avoid converting them ourselves, we need a vendor-neutral data provider, such as *Micrometer*.

## Micrometer

To solve this problem of being a vendor-neutral data provider, *Micrometer* came to be. It exposes Actuator metrics to external monitoring systems such as Prometheus, Netflix Atlas, AWS Cloudwatch (<https://aws.amazon.com/cloudwatch/>), and many more.

They quite correctly describe themselves as:

*Think SLF4J, but for metrics.*


Just as a refresher, SLF4J is a logging facade for other Java logging frameworks. SLF4J itself does not have any logging implementation. The idea is that you write code using SLF4J API's and the real implementation of it comes from the framework you choose. It could be any of the popular frameworks like log4j (<https://logging.apache.org/log4j/2.x/>), logback (<http://logback.qos.ch/>), etc.

Similarly, *Micrometer* automatically exposes `/actuator/metrics` data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application.

*Micrometer* is a separate open-sourced project and is not in the Spring ecosystem, so we have to explicitly add it as a dependency. Since we will be using *Prometheus*, let's add it's specific dependency in our `pom.xml`:

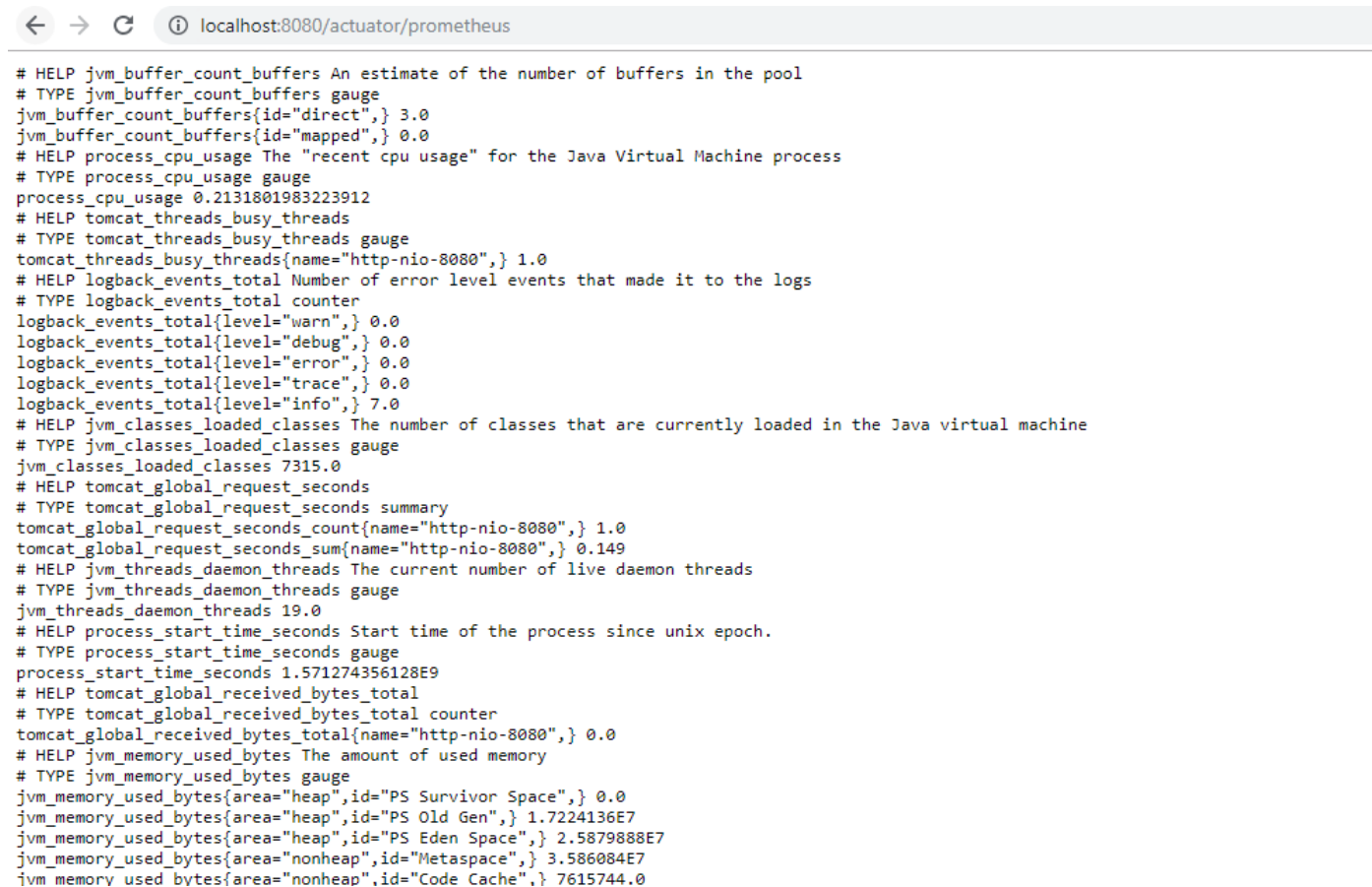
```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Rebuild and start the application and navigate our browser to  
<http://localhost:8080/actuator>:



```
{
  - _links: {
    + self: {...},
    + auditevents: {...},
    + beans: {...},
    + caches-cache: {...},
    + caches: {...},
    + health: {...},
    + health-component: {...},
    + health-component-instance: {...},
    + conditions: {...},
    + configprops: {...},
    + env: {...},
    + env-toMatch: {...},
    + info: {...},
    + loggers: {...},
    + loggers-name: {...},
    + heapdump: {...},
    + threaddump: {...},
    - prometheus: {
      href: "http://localhost:8080/actuator/prometheus",
      templated: false
    },
    + metrics: {...},
    + metrics-requiredMetricName: {...},
    + scheduledtasks: {...},
    + httptrace: {...},
    + mappings: {...}
  }
}
```

This will generate a new endpoint - `/actuator/prometheus`. Opening it, you will see data formatted specific for Prometheus:



```

← → ↻ ⓘ localhost:8080/actuator/prometheus

# HELP jvm_buffer_count_buffers An estimate of the number of buffers in the pool
# TYPE jvm_buffer_count_buffers gauge
jvm_buffer_count_buffers{id="direct"}, 3.0
jvm_buffer_count_buffers{id="mapped"}, 0.0
# HELP process_cpu_usage The "recent cpu usage" for the Java Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage 0.2131801983223912
# HELP tomcat_threads_busy_threads
# TYPE tomcat_threads_busy_threads gauge
tomcat_threads_busy_threads{name="http-nio-8080"}, 1.0
# HELP logback_events_total Number of error level events that made it to the logs
# TYPE logback_events_total counter
logback_events_total{level="warn"}, 0.0
logback_events_total{level="debug"}, 0.0
logback_events_total{level="error"}, 0.0
logback_events_total{level="trace"}, 0.0
logback_events_total{level="info"}, 7.0
# HELP jvm_classes_loaded_classes The number of classes that are currently loaded in the Java virtual machine
# TYPE jvm_classes_loaded_classes gauge
jvm_classes_loaded_classes 7315.0
# HELP tomcat_global_request_seconds
# TYPE tomcat_global_request_seconds summary
tomcat_global_request_seconds_count{name="http-nio-8080"}, 1.0
tomcat_global_request_seconds_sum{name="http-nio-8080"}, 0.149
# HELP jvm_threads_daemon_threads The current number of live daemon threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads 19.0
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.571274356128E9
# HELP tomcat_global_received_bytes_total
# TYPE tomcat_global_received_bytes_total counter
tomcat_global_received_bytes_total{name="http-nio-8080"}, 0.0
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="PS Survivor Space"}, 0.0
jvm_memory_used_bytes{area="heap",id="PS Old Gen"}, 1.7224136E7
jvm_memory_used_bytes{area="heap",id="PS Eden Space"}, 2.5879888E7
jvm_memory_used_bytes{area="nonheap",id="Metaspace"}, 3.586084E7
jvm_memory_used_bytes{area="nonheap",id="Code Cache"}, 7615744.0

```

# Prometheus



Prometheus is a time-series database that stores our metric data by pulling it (using a built-in data scraper) periodically over HTTP. The intervals between pulls can be configured, of course, and we have to provide the URL to pull from. It also has a simple user interface where we can visualize/query on all of the collected metrics.

Let's configure Prometheus, and more precisely the scrape interval, the targets, etc. To do that, we'll be using the `prometheus.yml` file:

```
global:
  scrape_interval: 10s

scrape_configs:
  - job_name: 'spring_micrometer'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['192.168.2.8:8080']
```

As you can see, we have a `scrape_configs` root key where we can define a list of jobs and specify the URL, metrics path, and the interval. If you'd like to read more about Prometheus configurations, feel free to visit the official documentation (<https://prometheus.io/docs/prometheus/latest/configuration/configuration/>).

**Note:** Since we are using Docker to run Prometheus, it will be running in a Docker network that won't understand `localhost/120.0.01`, as you might expect. Since our app is running on `localhost`, and for the Docker container, `localhost` means its own network, we have to specify our system IP in place of it.

## Subscribe to our Newsletter

Get occasional tutorials, guides, and reviews in your inbox. No spam ever.

Unsubscribe at any time.

Enter your email...

Subscribe

So instead of using `localhost:8080`, `192.168.2.8:8080` is used where `192.168.2.8` is my PC IP at the moment.

To check your system IP you can run `ipconfig` or `ifconfig` in your terminal, depending upon your OS.

Now, we can run Prometheus using the Docker command:

```
$ docker run -d -p 9090:9090 -v <path-to-prometheus.yml>:/etc/prometheus/prometheus.yml  
l prom/prometheus
```

`<path-to-prometheus.yml>` is where your own `prometheus.yml` is starting from the root. For an example, this works on my local Windows PC:

```
$ docker run -d -p 9090:9090 -v $PWD/prometheus.yml:/etc/prometheus/prometheus.yml pro  
m/prometheus
```

To see Prometheus dashboard, navigate your browser to `http://localhost:9090` (`http://localhost:9090`):

localhost:9090/graph

Prometheus Alerts Graph Status Help

☐ Enable query history

Expression (press Shift+Enter for newlines)

Execute - insert metric at cursor -

Graph Console

◀ Moment ▶

Element	Value
no data	

Add Graph

To check if Prometheus is actually listening to the Spring app, you can go to the `/targets` endpoint:

localhost:9090/targets

Prometheus Alerts Graph Status Help

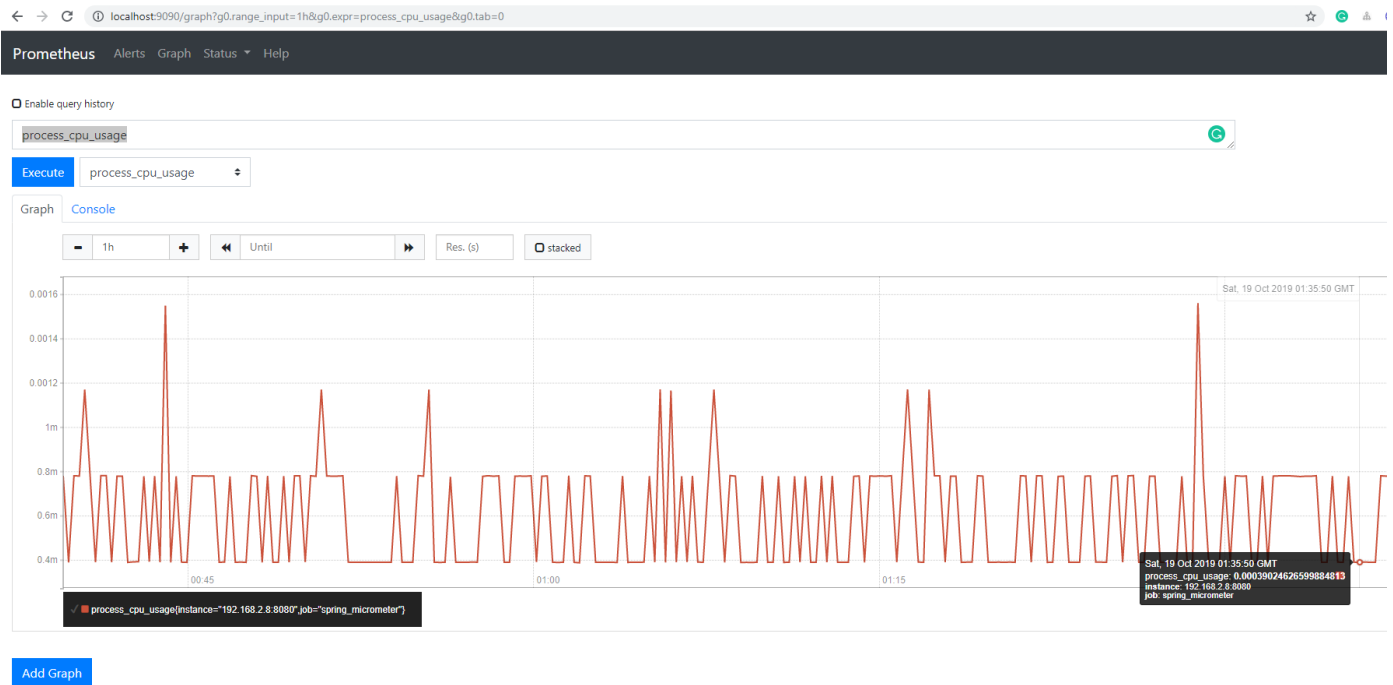
## Targets

All Unhealthy

spring\_micrometer (1/1 up) [show less](#)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://192.168.2.8:8080/actuator/prometheus">http://192.168.2.8:8080/actuator/prometheus</a>	UP	instance="192.168.2.8:8080" job="spring_micrometer"	2.89s ago	31.2ms	

Let's go back to the home page and select a metric from the list and click Execute:



## Prometheus Query Language - PromQL

Another thing to note is - Prometheus has its own query language called PromQL. It allows the user to select and aggregate time series data in real-time, storing it either in graph or tabular format. Alternatively, you can feed it to an external API through HTTP.

If you'd like to read more about PromQL, the official documentation (<https://prometheus.io/docs/prometheus/latest/querying/basics/>) covers it quite nicely.

## Grafana

While Prometheus does provide some crude visualization, Grafana offers a rich UI where you can build up custom graphs quickly and create a dashboard out of many graphs in no time. You can also import many community built dashboards for free and get going.

Grafana can pull data from various data sources like Prometheus, Elasticsearch (<https://www.elastic.co/>), InfluxDB (<https://www.influxdata.com/>), etc. It also allows you to set rule-based alerts, which then can notify you over Slack, Email, Hipchat, and similar.

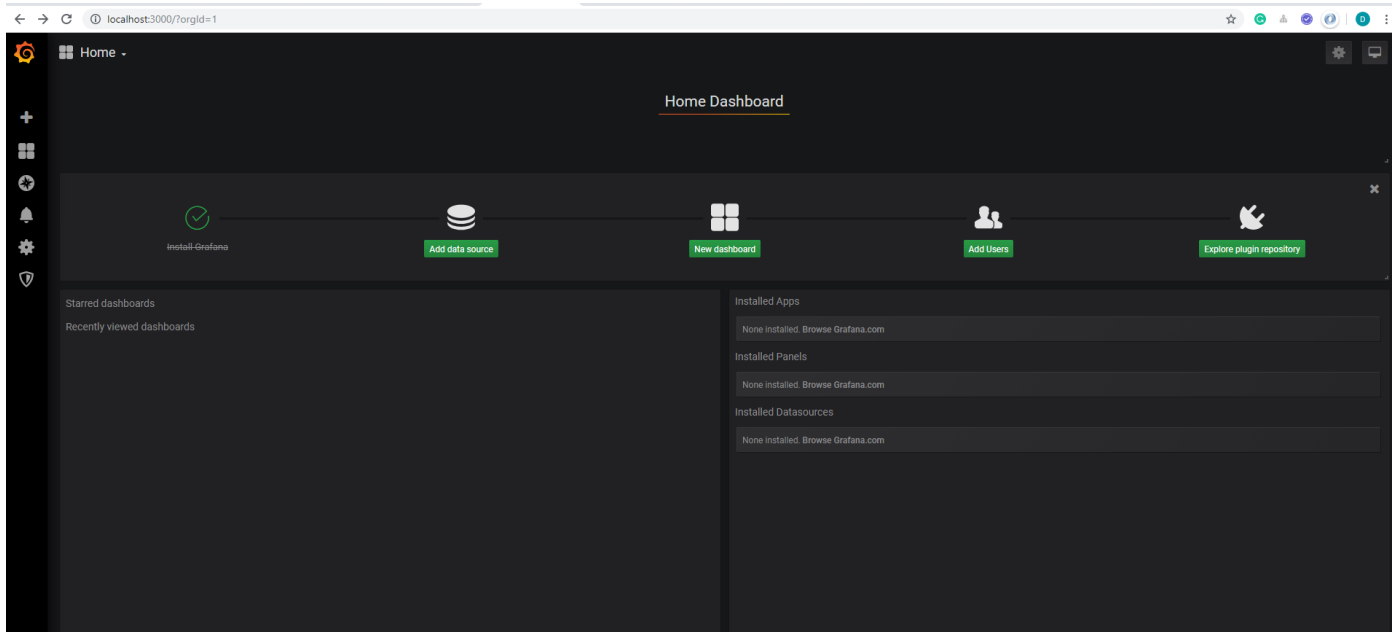
Let's start off by running Grafana using Docker:

```
$ docker run -d -p 3000:3000 grafana/grafana
```

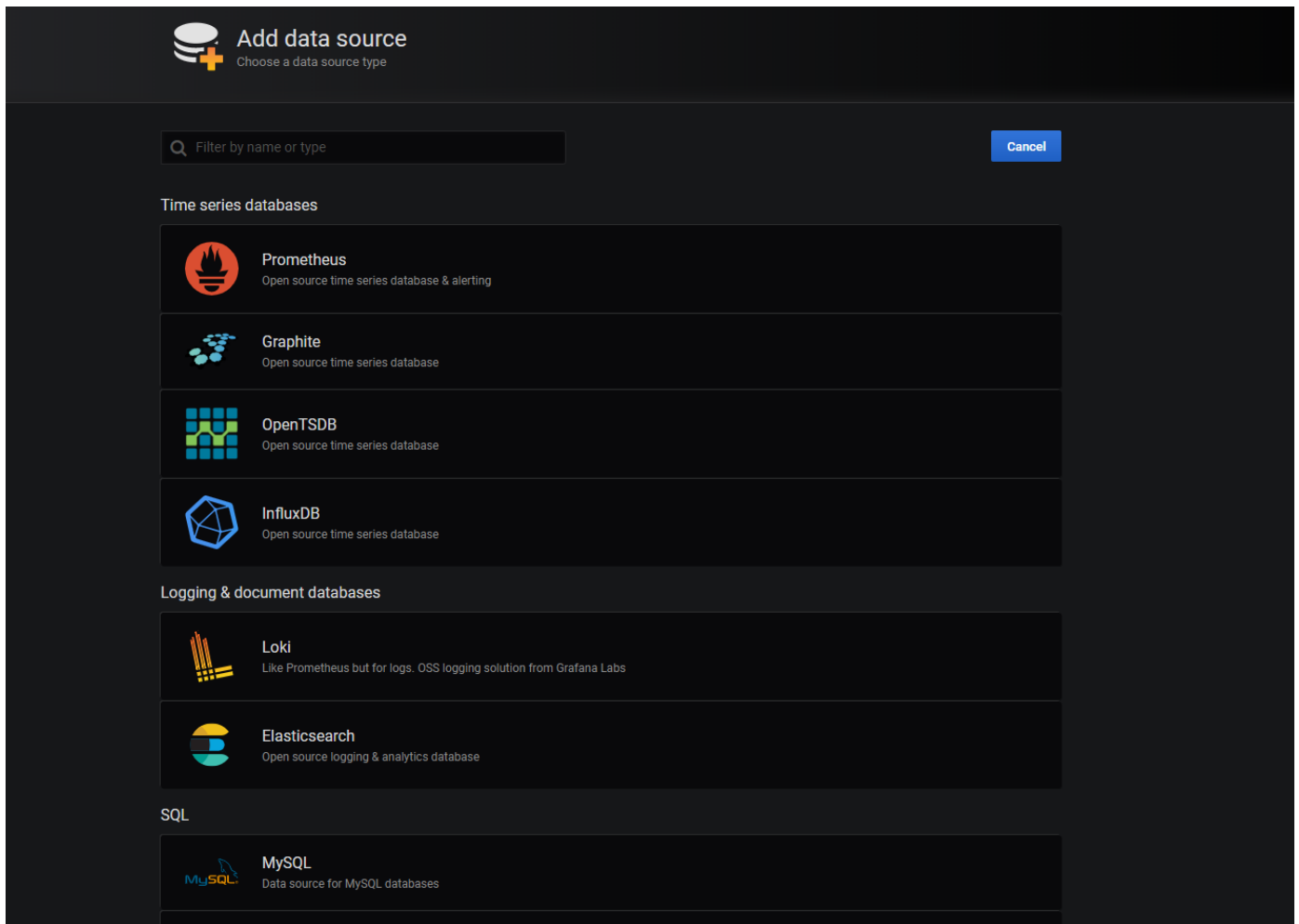
If you visit `http://localhost:3000`, you will be redirected to a login page:



The default username is `admin` and the default password is `admin`. You can change these in the next step, which is highly recommended:

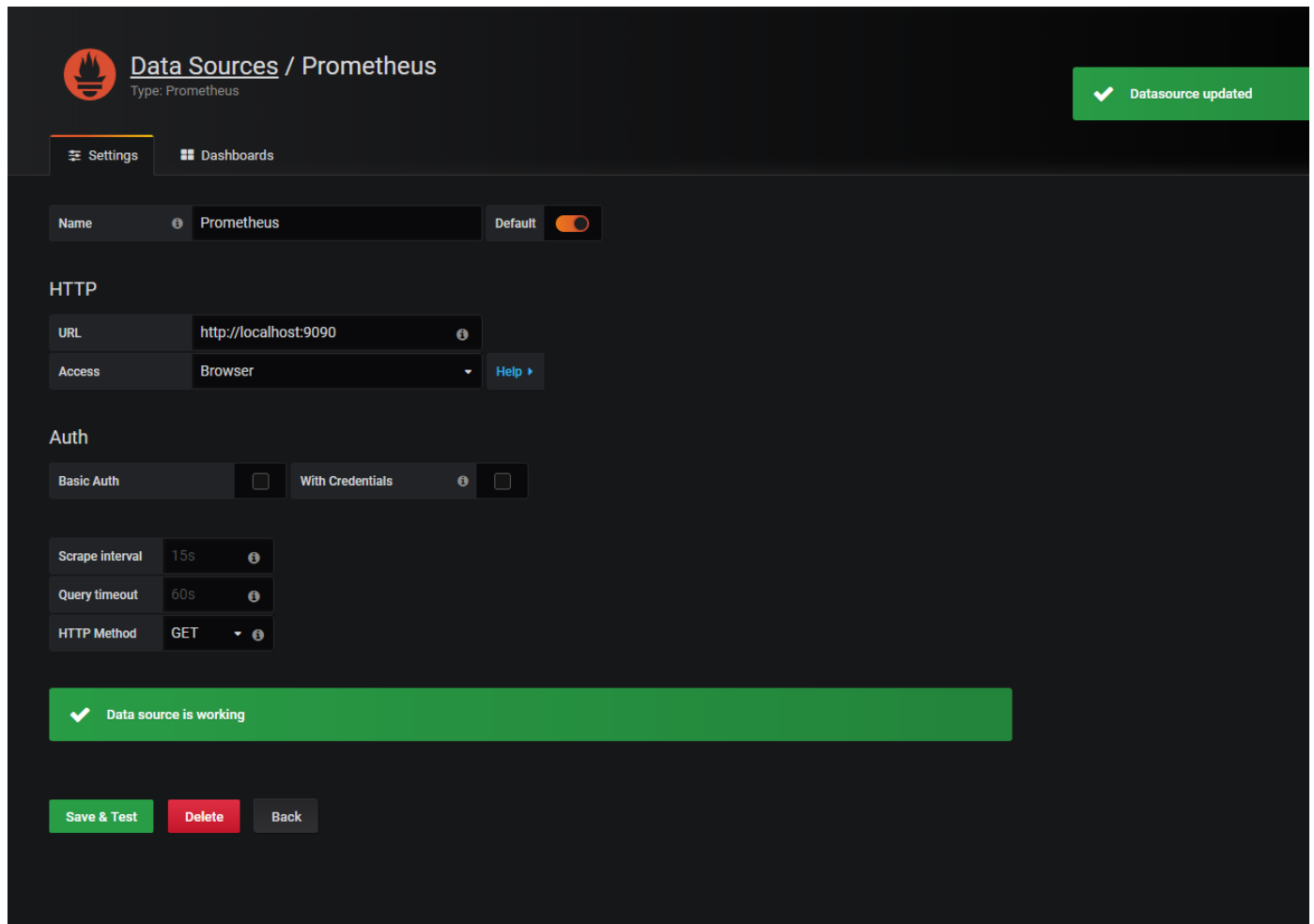


Since Grafana works with many data sources, we need to define which one we're relying on. Select Prometheus as your data source:



Now, add the URL that Prometheus is running on, in our case  
`http://localhost:9090` and select Access to be through a browser.

At this point, we can save and test to see if the data source is working correctly:



**Data Sources / Prometheus**  
Type: Prometheus

✓ Datasource updated

Settings Dashboards

Name Prometheus Default ☒

HTTP

URL http://localhost:9090 ⓘ

Access Browser Help ▶

Auth

Basic Auth ☐ With Credentials ⓘ ☐

Scrape interval 15s ⓘ

Query timeout 60s ⓘ

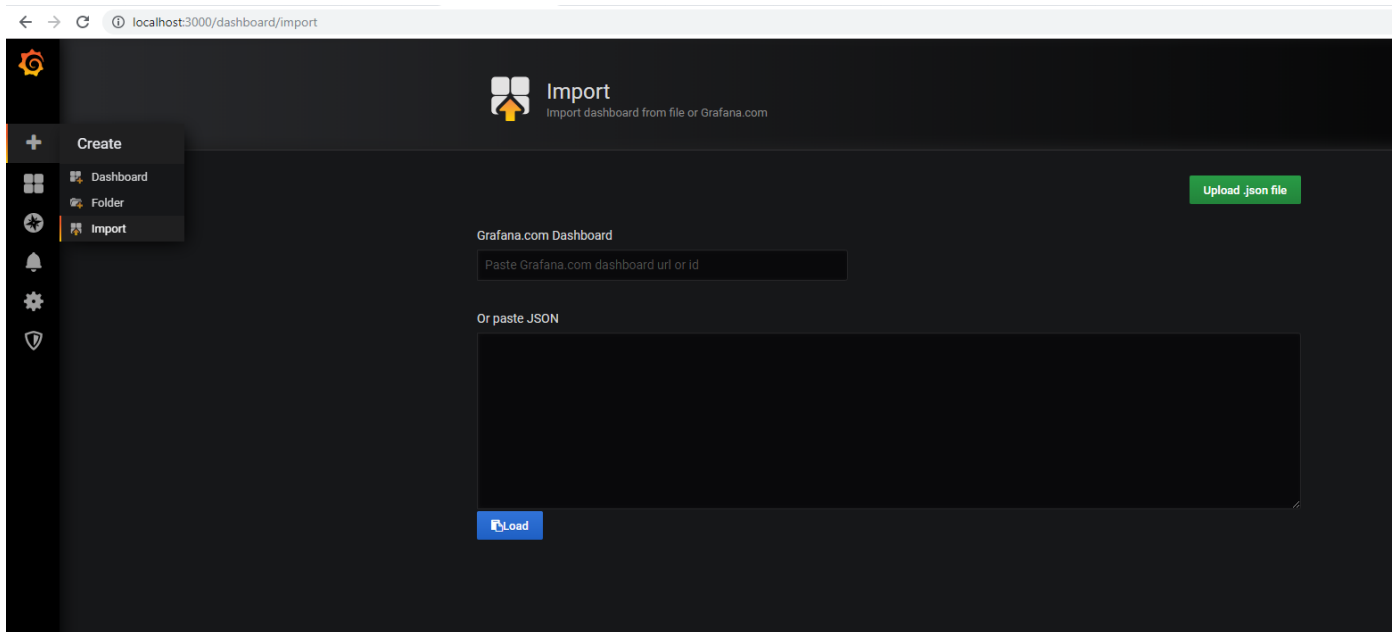
HTTP Method GET ⓘ

✓ Data source is working

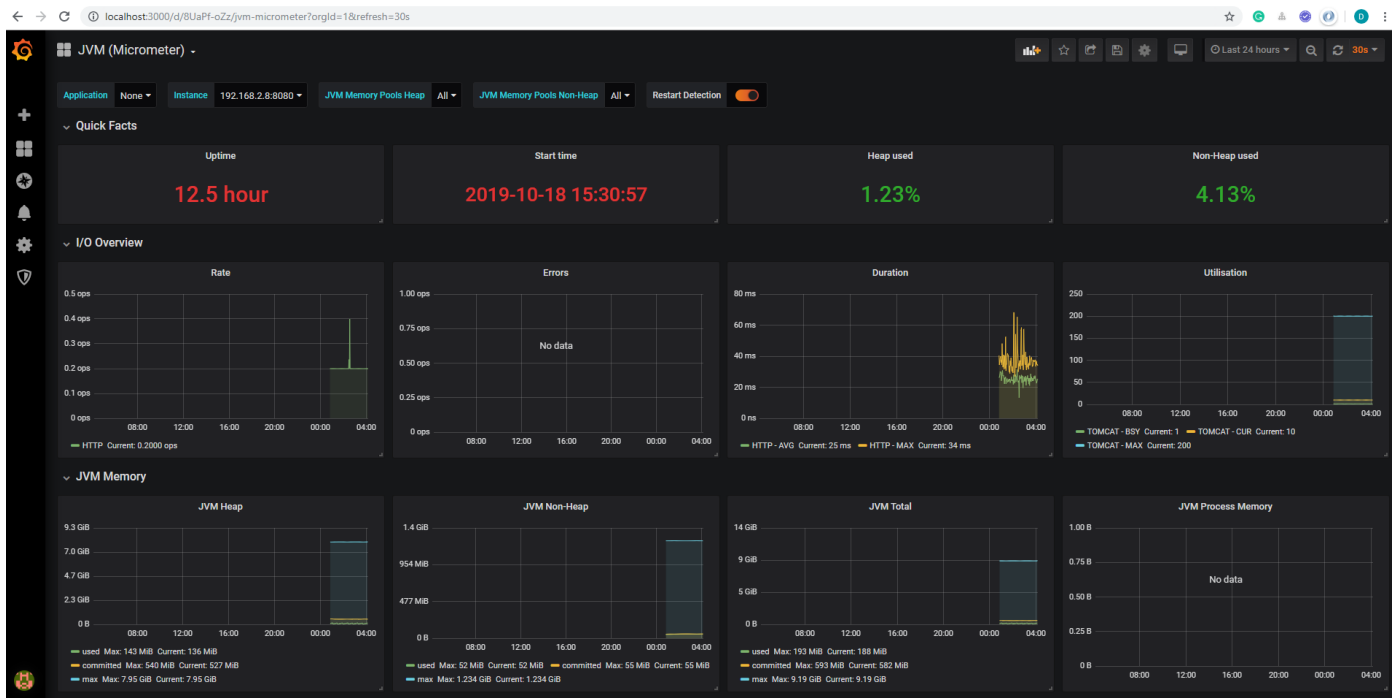
Save & Test Delete Back

As previously said, Grafana has a ton of pre-built dashboards (<https://grafana.com/grafana/dashboards>). For Spring Boot projects, the JVM dashboard (<https://grafana.com/grafana/dashboards/4701>) is popular:





Input the URL for the dashboard, select "Already created Prometheus datasource" and then click *Import*:



## Conclusion

Monitoring an application's health and metrics helps us manage it better, notice unoptimized behavior, and better understand its performance. This especially holds true when we're developing a system with many microservices, where monitoring each service can prove to be crucial when it comes to maintaining our system.

Based on this information, we can draw conclusions and decide which microservice needs to scale if further performance improvements can't be achieved with the current setup.

In this article, we used Micrometer to reformat the metrics data provided by Spring Boot Actuator and expose it in a new endpoint. This data was then regularly pulled and stored by Prometheus, which is a time-series database. Ultimately, we've used Grafana to visualize this information with a user-friendly dashboard.

As always, the code for the examples used in this article can be found on GitHub (<https://github.com/StackAbuse/microservices-monitoring>).

---

📁 [java \(/tag/java/\)](#), [spring \(/tag/spring/\)](#), [spring boot \(/tag/spring-boot/\)](#)

[62C%20Prometheus%2C%20and%20Grafana&url=https://stackabuse.com/monitoring-om/monitoring-spring-boot-apps-with-micrometer-prometheus-and-grafana/|spring-boot-apps-with-micrometer-prometheus-and-grafana/stackabuse.com/monitoring-spring-boot-apps-with-micrometer-prometheus-and-](#)



[\(/author/dhananjay/\)](#)

About Dhananjay Singh [\(/author/dhananjay/\)](#)

## Subscribe to our Newsletter

Get occasional tutorials, guides, and reviews in your inbox. No spam ever. Unsubscribe at any time.

0 Comments

StackAbuse

 ravi reddy ▾ Recommend Tweet Share

Sort by Best ▾

StackAbuse requires you to verify your email address before posting. Send verification email to ienroll@gmail.com ✕



Use PC IP when configuring Prometheus Datasource!

Be the first to comment.

#### ALSO ON STACKABUSE

### Deploying a Flask Application to Heroku

1 comment • 2 months ago



**Todd Flanders** — It seems the build pack wants Procfile, not Procfile.txt

### Basics of Memory Management in Python

1 comment • 3 months ago



**AlbertMietus** — Remember: garbage collection (at least the implementation) is part of the compiler; not the language!Ref-

### Image Classification with Transfer Learning and PyTorch

1 comment • 3 months ago



**akhila** — HOW to do transfer learning for grayscale images?

### Uploading Files to AWS S3 with Python and Django

1 comment • a month ago



**Eti Uthakima.** — kazi poa!

< Previous Post : Introduction to PyTorch for Classification (/introduction-to-pytorch-for-classification/)

g File Uploads in Node.js with Express and Multer > (/handling-file-uploads-in-node-js-with-express-and-multer/)

## Ad

The logo for CashNetUSA, with 'CashNet' in orange and 'USA' in blue, followed by a registered trademark symbol.

One step left  
to go on your  
application!

Finish Today

## Follow Us

-  Twitter (<https://twitter.com/StackAbuse>)
-  Facebook (<https://www.facebook.com/stackabuse>)
-  RSS (<https://stackabuse.com/rss/>)

## Newsletter

Subscribe to our newsletter! Get occasional tutorials, guides, and reviews in your inbox.

No spam ever. Unsubscribe at any time.

## Ad

### FEATURED VIDEOS

Powered by **[primis]**



### Largest Starbucks Store Ever Opens in...

Largest Starbucks Store Ever Debuts in Chicago. The Starbucks Roastery, which is the coffee's giant's...

## Our Sponsors



(<https://stackabu.se/digitalocean>)

The simplest cloud platform for  
developers and teams.

Learn More (<https://stackabu.se/digitalocean>)

## Want a remote job?

Code Challenge Reviewer

**Geektastic** 2 days ago (<https://horeremote.io/remote->

job/0696-code-challenge-reviewer-at-geektastic)  
(<https://hiredremote.io/remote-job/0696-code-challenge-reviewer-at-geektastic>) `java`  
(<https://hiredremote.io/remote-java-jobs>) `python`  
(<https://hiredremote.io/remote-python-jobs>)  
`javascript` (<https://hiredremote.io/remote-javascript-jobs>)  
`ruby` (<https://hiredremote.io/remote-ruby-jobs>)

---

#### Software Engineer

##### **Achievement Network** 4 days ago

(<https://hiredremote.io/remote-job/0279-software-engineer-at-achievement-network>)  
(<https://hiredremote.io/remote-job/0279-software-engineer-at-achievement-network>) `java`  
(<https://hiredremote.io/remote-java-jobs>) `javascript`  
(<https://hiredremote.io/remote-javascript-jobs>) `mysql`  
(<https://hiredremote.io/remote-mysql-jobs>) `aws`  
(<https://hiredremote.io/remote-aws-jobs>)

---

#### Application Security Engineer

##### **New Context Services** 4 days ago

(<https://hiredremote.io/remote-job/0444-application-security-engineer-at-new-context-services>)  
(<https://hiredremote.io/remote-job/0444-application-security-engineer-at-new-context-services>) `aws`  
(<https://hiredremote.io/remote-aws-jobs>) `java`  
(<https://hiredremote.io/remote-java-jobs>) `python`  
(<https://hiredremote.io/remote-python-jobs>) `ruby`  
(<https://hiredremote.io/remote-ruby-jobs>)

 [More jobs \(https://hiredremote.io\)](https://hiredremote.io)

Jobs via HireRemote.io (<https://hiredremote.io>)

---

## Recent Posts

[Message Queueing in Node.js with AWS SQS \(/message-queueing-in-node-js-with-aws-sqs/\)](/message-queueing-in-node-js-with-aws-sqs/)

---

[Introduction to JavaScript Proxies in ES6 \(/introduction-to-javascript-proxies-in-es6/\)](/introduction-to-javascript-proxies-in-es6/)

---

[Deploying a Node.js App to a DigitalOcean Droplet with Docker \(/deploying-a-node-js-app-to-a-digitalocean-droplet-with-docker/\)](/deploying-a-node-js-app-to-a-digitalocean-droplet-with-docker/)

---

## Tags

---

[ai \(/tag/ai/\)](/tag/ai/)[algorithms \(/tag/algorithms/\)](/tag/algorithms/)[amqp \(/tag/amqp/\)](/tag/amqp/)[angular \(/tag/angular/\)](/tag/angular/)[announcements \(/tag/announcements/\)](/tag/announcements/)[apache \(/tag/apache/\)](/tag/apache/)[arduino \(/tag/arduino/\)](/tag/arduino/)[artificial intelligence \(/tag/artificial-intelligence/\)](/tag/artificial-intelligence/)[asynchronous \(/tag/asynchronous/\)](/tag/asynchronous/)[aws \(/tag/aws/\)](/tag/aws/)

## Follow Us

---

 [Twitter \(https://twitter.com/StackAbuse\)](https://twitter.com/StackAbuse)

 [Facebook \(https://www.facebook.com/stackabuse\)](https://www.facebook.com/stackabuse)

 [RSS \(https://stackabuse.com/rss/\)](https://stackabuse.com/rss/)

---

Copyright © 2019, Stack Abuse (<https://stackabuse.com>). All Rights Reserved.

[Disclosure \(/disclosure\)](/disclosure) • [Privacy Policy \(/privacy-policy\)](/privacy-policy) • [Terms of Service \(/terms-of-service\)](/terms-of-service)