The dependencies are defined in the sample02/pom.xml file:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

To enable exposure of the metrics at runtime, the following properties need to be enabled in the sample02/src/resources/application.properties file:

management.endpoints.web.exposure.include=prometheus,info,health

management.endpoint.prometheus.enabled=true

Now that we have our Order Processing microservice exposing the required metrics, the next step is to configure and start Prometheus so that it can read the exposed metrics. In this particular example, we have created a docker-compose script that first starts the Prometheus container and then the Grafana container. To run these containers, navigate to the sample02/monitoring directory by using your command-line client and execute the following command:

\> docker-compose -f docker-compose.yml up

If this is the first time you are starting these containers, it might take a few minutes for startup because it downloads the container images from Docker Hub (<https://hub.docker.com/>) and copies to your local Docker registry. These containers would start up in a matter of seconds in subsequent attempts. Once the containers have started successfully, you should see two messages as follows:

Starting prometheus ... done

Starting grafana ... done

To see Prometheus in action, open a new tab in your web browser and navigate to http://localhost:9090. From the top menu, click the Status drop-down list and choose Targets. You should be able to see a target named order-processing with its state as UP. You should see the last scrape timestamp information as well. This means that Prometheus is able to read the metrics exposed by our Spring Boot service.

Next, click the Graph link from the top menu. This UI in Prometheus allows you to query various metrics if you know their names. To check how much memory the JVM consumes, enter the string jvm\_memory\_used\_bytes in the provided text box and click the Execute button. The Graph tab gives you a view of the memory consumed over a period of time, and the Console tab shows you the exact values at that particular moment. The Graph view looks similar to [figure 5.10](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/section-52-39.xhtml#fig5-10).

Figure 5.10: Graph view of the jvm\_memory\_used\_bytes metric as displayed in the Prometheus UI

Larger ViewGraphical user interface, text, application

Description automatically generated

To understand how Prometheus scrapes the Order Processing microservice for information, you can open the monitoring/prometheus/prometheus.yml file. This is the Prometheus configuration file. The scrape configurations shown in the following listing help Prometheus find the Order Processing microservice and its metrics.

**Listing 5.3: The Prometheus scrape configuration**

scrape\_configs:

- job\_name: 'order-processing'

scrape\_interval: 10s ❶

metrics\_path: '/actuator/prometheus' ❷

static\_configs:

- targets: ['host.docker.internal:8080'] ❸

labels:

application: 'order-processing-app'

*❶ Defines the frequency at which metrics should be collected from each target under the order-processing job*

*❷ Defines the path under which the metrics of the Order Processing microservice are hosted*

*❸ Specifies the hosts that require monitoring under this job*

As you have noticed from using the Prometheus UI, we need to know the parameters to watch out for in order to use the default Prometheus UI for monitoring our microservices. In terms of having an overall view of the state of a microservices deployment, this experience/process does not help us a lot. This is where Grafana comes into the picture and helps us build dashboards for an overall view. Let's take a look at creating a dashboard in Grafana for a better view.

By now, our Grafana container is up and running. We can therefore use it directly. To do that, open a new tab in your browser console and navigate to http://localhost:3000. Enter admin as the username, and password as the password to log in to Grafana.

Next, we need to install a dashboard on Grafana. To do that, hover over the Create menu item on the left menu panel (it should be the first item in the menu) and click the Import link. In the page that appears, click the Upload .JSON File button and choose to upload the sample02/monitoring/grafana/dashboard.json file.

In the form that appears next, go with the defaults for all the fields except the Prometheus field, where you are expected to select a data source for this dashboard. Select Prometheus as the data source for its value and proceed to import this dashboard.

Next, under Dashboards in the left menu pane, click the Manage link. You should see a dashboard named JVM (Micrometer). Once you click this dashboard, you should see widgets being loaded onto the UI. They are categorized into sections such as Quick Facts, I/O Overview, and JVM Memory. At first, it might take a short while for the widgets to load. After they are loaded, you should see something similar to [figure 5.11](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/section-52-39.xhtml#fig5-11).

Figure 5.11: The Grafana dashboard for our microservice. The metrics exposed by our microservice are scraped by Prometheus periodically. Grafana queries Prometheus by using PromQL and visualizes the metrics

Larger ViewA screenshot of a computer

Description automatically generated with medium confidence

As you can see, Grafana gives you a much more user-friendly view of the metrics exposed by the Order Processing microservice. To understand how Grafana queries data from Prometheus, you need to take a look at the sample02/monitoring/grafana/provisioning/datasources/datasource.yml file. This file contains the Prometheus URL so that Grafana can connect to it and query its data. The dashboard.json file located in the sample02/monitoring/grafana directory defines, in JSON format, the type of metrics to be visualized under each widget. For example, take a look at the following JSON, which visualizes the Uptime panel on this dashboard:

"targets": [

{

"expr": "process\_uptime\_seconds{application=\"$application\",

instance=\"$instance\"}",

"format": "time\_series",

"intervalFactor": 2,

"legendFormat": "",

"metric": "",

"refId": "A",

"step": 14400

}

],

"thresholds": "",

"title": "Uptime",

We now have a bit of experience in using Prometheus to scrape metrics from our exposed microservice and in using Grafana to visualize these metrics.

**5.2.2 Behind the Scenes of Using Prometheus for Monitoring**

Prometheus, an open source tool for system monitoring and alerting, is a standalone project maintained by the community itself. It is part of the Cloud Native Computing Foundation (CNCF) and the second hosted project in CNCF. As of this writing, only ten projects have graduated in CNCF, and Prometheus is one of them.

Prometheus is also the most popular open source monitoring tool available. When using Prometheus to monitor a microservices deployment, it's important to understand a few things regarding how Prometheus works.

**SCRAPING DATA FROM MICROSERVICES TO MONITOR**

Prometheus pulls metrics data from microservices on a periodic time interval. As you've learned, this is known as *scraping*. Each microservice needs to have an exposed endpoint, which contains details about the various metrics we need to monitor. The Prometheus server connects to these endpoints periodically and pulls down the information it needs for its monitoring purposes.

Prometheus also has a push-gateway for supporting short-lived processes. Processes that may not live long enough for Prometheus to scrape can push their metrics to a push gateway before dying off. The push gateway acts as a metrics cache for the processes that no longer exist.

**WHAT IS TIME-SERIES DATA?**

Prometheus stores metrics in a time-series database at millisecond precision. A time-series database contains a recording of various metrics against the time at which it was recorded. This data is stored for a period of time and is usually presented in line graphs against time.

**DEFINING A METRIC IN PROMETHEUS**

A *metric* in Prometheus is an immutable block of data identified using both the metric name and labels. A metric is stored against its timestamp. Given a metric name and labels, time series are identified using the following notion:

<metric\_name>={<label\_name>=<label\_value>, ... .}

For example, a metric used for getting the total number of HTTP requests would look like the following:

http\_requests\_total={method="POST", path="/menu", type="JSON"}

[Figure 5.12](https://cdn2.percipio.com/1639251572.d6a6f32c7fbca9f0e707995eb0f2a271f86a62e7/eod/books/154188/OEBPS/section-52-39.xhtml#fig5-12) illustrates the architecture of Prometheus.

Figure 5.12: A Prometheus server scrapes the microservices and push gateway for metrics. It then uses these metrics to trigger alerts. The Prometheus web UI and Grafana use PromQL to retrieve data from Prometheus for visualization

Larger ViewDiagram

Description automatically generated

As you can see, each microservice needs to expose an endpoint from which Prometheus scrapes metrics information. When exposing this endpoint, we need to ensure that this endpoint is secured, using TLS so that the information passed on the wire is kept safe from intruders, and using an authentication mechanism such as OAuth 2.0 or basic authentication.