**Working with Kubernetes Monitoring**

**Overview**

In this series of labs, you take a demo microservices Java application built with the Spring framework and modify it to use an external database server. You adopt some of the best practices for tracing, configuration management, and integration with other services using integration patterns.

In the previous lab, you containerized the application and deployed it to a Kubernetes Engine cluster with Cloud Kubernetes Monitoring support. That means you can monitor the health of the Kubernetes Engine cluster using Cloud Monitoring. A replica of that environment is preconfigured for you in this lab.

Traditionally, Java applications are monitored through JMX metrics, which provide metrics on such things as thread count and heap usage. In the cloud-native world where you monitor more than just the Java stack, you need to use more generic metrics formats, such as Prometheus.

Cloud Kubernetes Monitoring aggregates logs, events, and metrics from your Kubernetes Engine environment to help you understand your application's behavior in production. Prometheus is an optional monitoring tool often used with Kubernetes. If you configure Cloud Kubernetes Monitoring with Prometheus support, then services that expose metrics using the Prometheus data model also have their data exported from the cluster and made visible as external metrics in Cloud Monitoring.

In this lab, you enable Prometheus monitoring for Kubernetes and then modify the demo application to expose Prometheus metrics from within the application and its backend service. You can then use Cloud Monitoring to monitor internal performance metrics from your application while it is running on Kubernetes Engine.

**Objectives**

In this lab, you learn how to perform the following tasks:

* Enable Cloud Monitoring for Kubernetes Engine
* Enable Prometheus monitoring in a Kubernetes Engine cluster
* Expose Prometheus metrics from inside a Spring Boot application
* Explore live application metrics using Cloud Monitoring

**Task 0. Lab Preparation**

**Access Qwiklabs**

**How to start your lab and sign in to the Console**

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



1. Copy the username, and then click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Choose an account** page.

***Tip:*** Open the tabs in separate windows, side-by-side.

1. On the Choose an account page, click **Use Another Account**.



1. The Sign in page opens. Paste the username that you copied from the Connection Details panel. Then copy and paste the password.

***Important:*** You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own GCP account, do not use it for this lab (avoids incurring charges).

1. Click through the subsequent pages:
   * Accept the terms and conditions.
   * Do not add recovery options or two-factor authentication (because this is a temporary account).
   * Do not sign up for free trials.

After a few moments, the GCP console opens in this tab.

**Note:** You can view the menu with a list of GCP Products and Services by clicking the **Navigation menu** at the top-left, next to “Google Cloud Platform”. 

After you complete the initial sign-in steps, the project dashboard appears.



**Fetch the application source files**

To begin the lab, click the **Activate Cloud Shell** button at the top of the Google Cloud Console. To activate the code editor, click the Open Editor button on the toolbar of the Cloud Shell window. This sets up the editor in a new tab with continued access to Cloud Shell.

**Note:** the lab setup includes automated deployment of the services that you configured yourself in previous labs. When the setup is complete, copies of the demo application (configured so that they are ready for this lab session) are put into a Cloud Storage bucket named using the project ID for this lab.

1. In the Cloud Shell command line, enter the following command to create an environment variable that contains the project ID for this lab:

export PROJECT\_ID=$(gcloud config list --format 'value(core.project)')

1. Verify that the demo application files were created.

gsutil ls gs://$PROJECT\_ID

If you get a BucketNotFound error, this means that the lab's deployment script has not finished yet. You will need to wait for the DM template to complete before proceeding. This usually takes around 10 minutes upon starting the lab. Please wait a few minutes then retry.

1. Copy the application folders to Cloud Shell.

gsutil -m cp -r gs://$PROJECT\_ID/\* ~/

1. Make the Maven wrapper scripts executable.

chmod +x ~/guestbook-frontend/mvnw

chmod +x ~/guestbook-service/mvnw

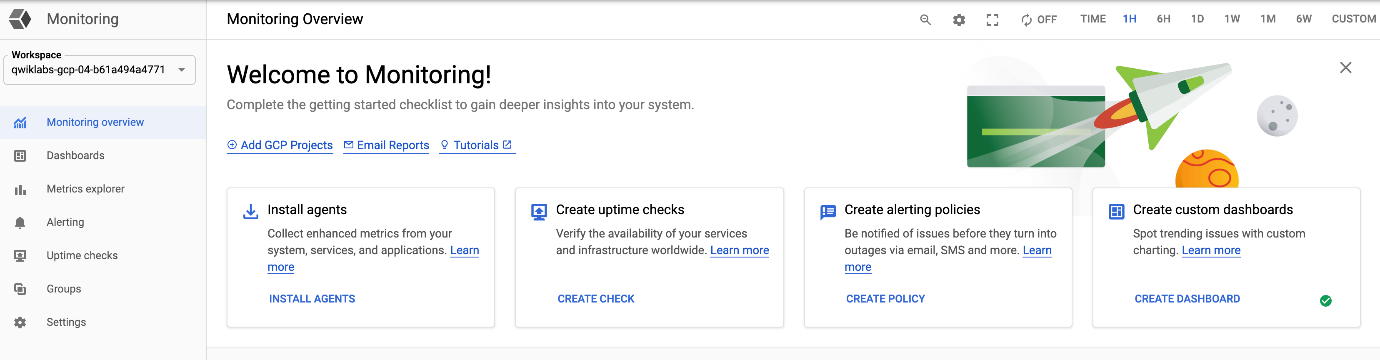
**Task 1. Enable Cloud Monitoring and view the Cloud Kubernetes Monitoring dashboard**

Create a Monitoring workspace

You will now setup a Monitoring workspace that's tied to your Qwiklabs GCP Project. The following steps create a new account that has a free trial of Monitoring.

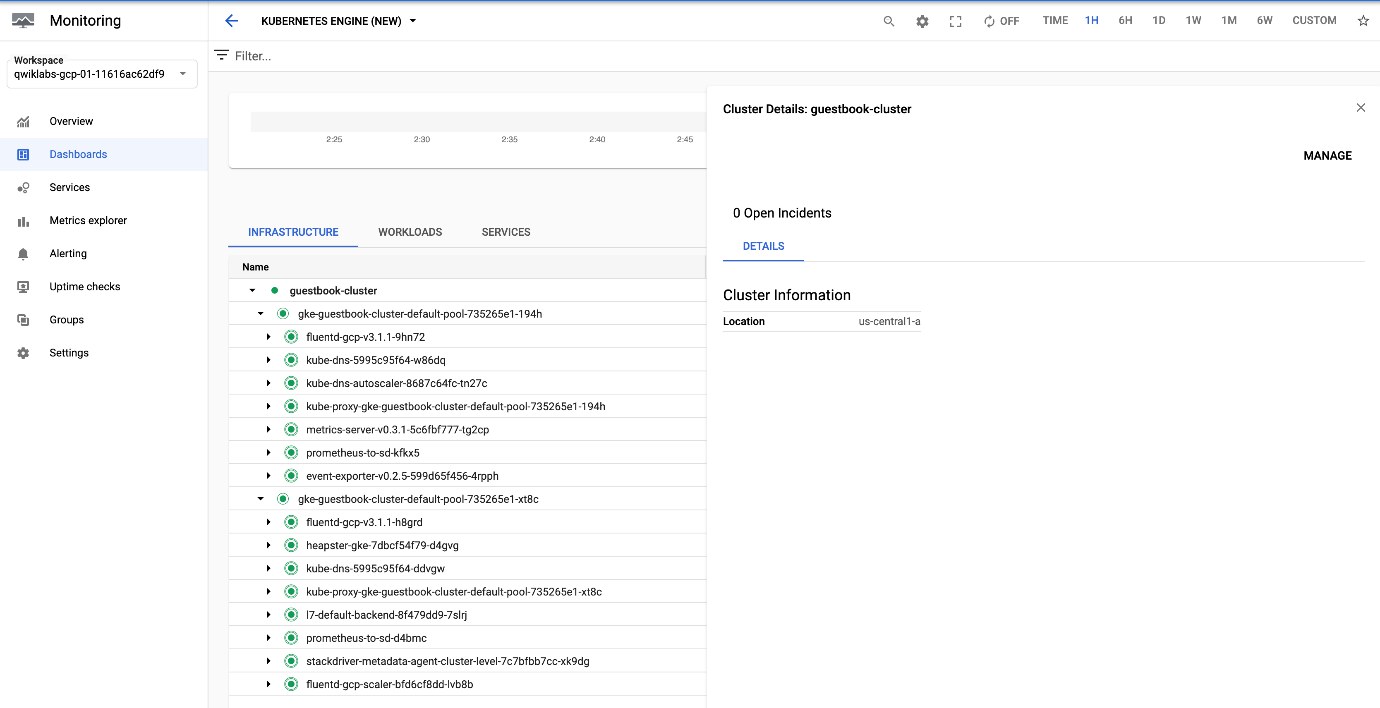
1. In the Google Cloud Platform Console, click on **Navigation menu** > **Monitoring**.
2. Wait for your workspace to be provisioned.

When the Monitoring dashboard opens, your workspace is ready.



1. Click **Dashboards** and select **Kubernetes Engine (NEW)** to view the Kubernetes monitoring dashboard. Click **guestbook-cluster**.

You may need to wait for a few minutes for the Kubernetes Engine cluster and its resources to become visible to Cloud Monitoring.



**Task 2. Expose Prometheus metrics from Spring Boot applications**

Spring Boot can expose metrics information through Spring Boot Actuator. Micrometer is the metrics collection facility included in Spring Boot Actuator. Micrometer can expose all the metrics using the Prometheus format.

If you are not using Spring Boot, you can expose JMX metrics through Prometheus by using a [Prometheus JMX Exporter agent](https://github.com/prometheus/jmx_exporter).

In this task, you add the Spring Boot Actuator starter and Micrometer dependencies to the guestbook frontend application.

1. In the Cloud Shell code editor, open ~/guestbook-frontend/pom.xml.
2. Insert the following new dependencies at the end of the <dependencies> section, just before the closing </dependencies> tag.

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.micrometer</groupId>

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

<scope>runtime</scope>

</dependency>

1. In the Cloud Shell code editor, open ~/guestbook-frontend/src/main/resources/application.properties.
2. Add the following two properties to configure Spring Boot Actuator to expose metrics on port 9000.

management.server.port=9000

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

1. To send log entries to Stackdriver Logging, via STDOUT and structured JSON logging, change guestbook-frontend/src/main/resources/logback-spring.xml to use the CONSOLE\_JSON appender. Copy and replace the entire contents of the file with the following code:

<configuration>

<include resource="org/springframework/boot/logging/logback/defaults.xml" />

<include resource="org/springframework/boot/logging/logback/console-appender.xml" />

<springProfile name="cloud">

<include resource="org/springframework/cloud/gcp/logging/logback-json-appender.xml"/>

<root level="INFO">

<appender-ref ref="CONSOLE\_JSON"/>

</root>

</springProfile>

<springProfile name="default">

<root level="INFO">

<appender-ref ref="CONSOLE"/>

</root>

</springProfile>

</configuration>

**Task 3. Rebuild the containers**

In this task you rebuild the containers and configure the frontend contaner deployment to expose the prometheus monitoring endpoint.

1. In the Cloud Shell change to the frontend application directory.

cd ~/guestbook-frontend

1. Build the frontend application container.

./mvnw clean compile jib:build

1. While this is compiling switch back to the Cloud Shell code editor and open ~/kustomize/base/guestbook-frontend-deployment.yaml.

You update the Kubernetes deployment to specify the Prometheus metrics endpoint. With this configuration, Spring Boot Actuator exposes the Prometheus metrics on port 9000, under the path of /actuator/prometheus.

1. Update the Kubernetes manifest to declare the metrics ports. You will be putting this under the kind: Deployment in the containers section.

- name: metrics

containerPort: 9000

The guestbook-frontend-deployment.yaml file should now look like the following screenshot:



**Note:** Whitespace is significant in YAML file layouts. The layout of the changes you make must match the screenshot.

1. When the frontend application build has completed in the Cloud Shell change to the guestbook backend service application directory.

cd ~/guestbook-service

1. Build the guestbook backend service application container.

./mvnw clean compile jib:build

**Note:** You haven't made any changes to the backend service application but you have to build the image so that it is available on the gcr.io container repository for the lab when you deploy the full application.

1. Redeploy the manifest:

mkdir -p ~/bin

cd ~/bin

curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install\_kustomize.sh" | bash

export PATH=$PATH:$HOME/bin

cd ~/kustomize/base

cp ~/service-account.json ~/kustomize/base

kustomize build

gcloud container clusters get-credentials guestbook-cluster --zone=us-central1-a

kustomize edit set namespace default

kustomize build | kubectl apply -f -

1. Wait for the pods to restart. Find the pod name for one of the instances.

kubectl get pods -l app=guestbook-frontend

You should see something like the following:

NAME READY STATUS RESTARTS AGE

guestbook-frontend-8567fdc8c8-c68vk 1/1 Running 0 5m

guestbook-frontend-8567fdc8c8-gvcf5 1/1 Running 0 5m

1. Establish a port forward to one of the Guestbook Frontend pod. Replacing [podnumber] with one of the ID's of the pods you received from the previous command:

kubectl port-forward guestbook-frontend-[podnumber] 9000:9000

**Task 4. Install Prometheus and Stackdriver Sidecar**

Stackdriver Kubernetes Monitoring can [monitor Prometheus metrics](https://cloud.google.com/stackdriver/docs/solutions/kubernetes-engine/prometheus) from the Kubernetes cluster. Install Prometheus support to the cluster.

1. In a **new** Cloud Shell tab, install a quickstart Prometheus operator.

gcloud container clusters get-credentials guestbook-cluster --zone=us-central1-a

kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/v0.38.1/bundle.yaml

1. Provision Prometheus using the Prometheus Operator.

cd ~/prometheus

export PROJECT\_ID=$(gcloud config list --format 'value(core.project)')

# Make sure the project ID is set

echo $PROJECT\_ID

cat prometheus.yaml | envsubst | kubectl apply -f -

kubectl apply -f pod-monitors.yaml

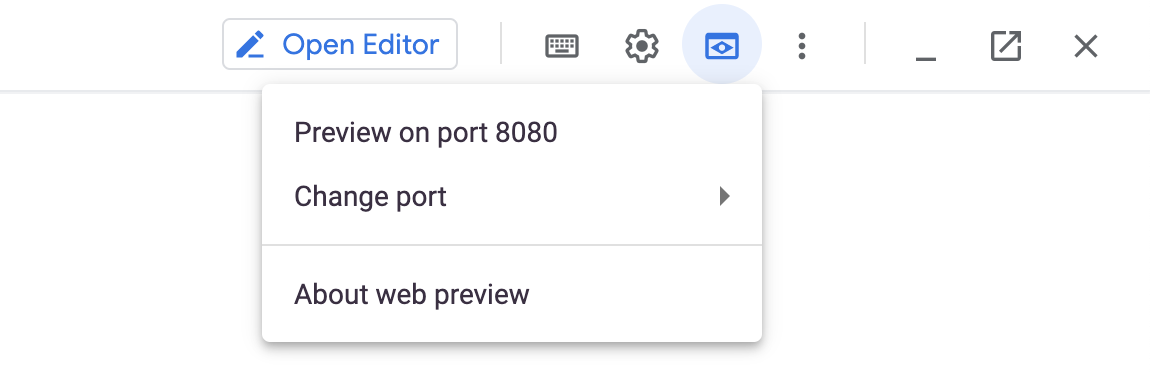
The prometheus.yaml file has an additional Stackdriver Sidecar that's designed to export the scraped Prometheus metrics to Stackdriver.

1. Validate Prometheus is running properly and scraping the data. Establish a port forward to Prometheus' port.

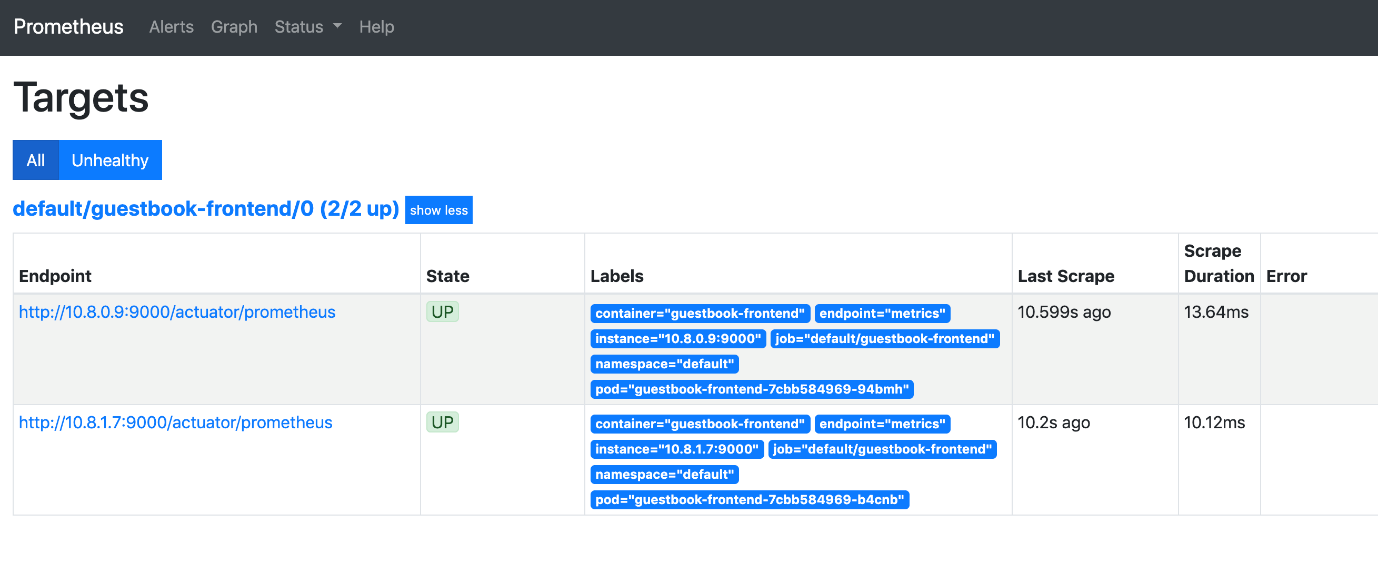
pkill java

kubectl port-forward svc/prometheus 9090:9090

1. Click Web Preview in the Cloud Shell, then click Preview on port 8080. It should open up a new page.



1. Now, in the URL, change the beginning of the line from 8080 to 9090 and refresh the page. Your URL should now look something like: https://9090-dot-12909153-dot-devshell.appspot.com/graph.
2. In the Prometheus console, select **Status** → **Targets**.
3. Observe that there are 2 targets (2 pods) being scraped for metrics.

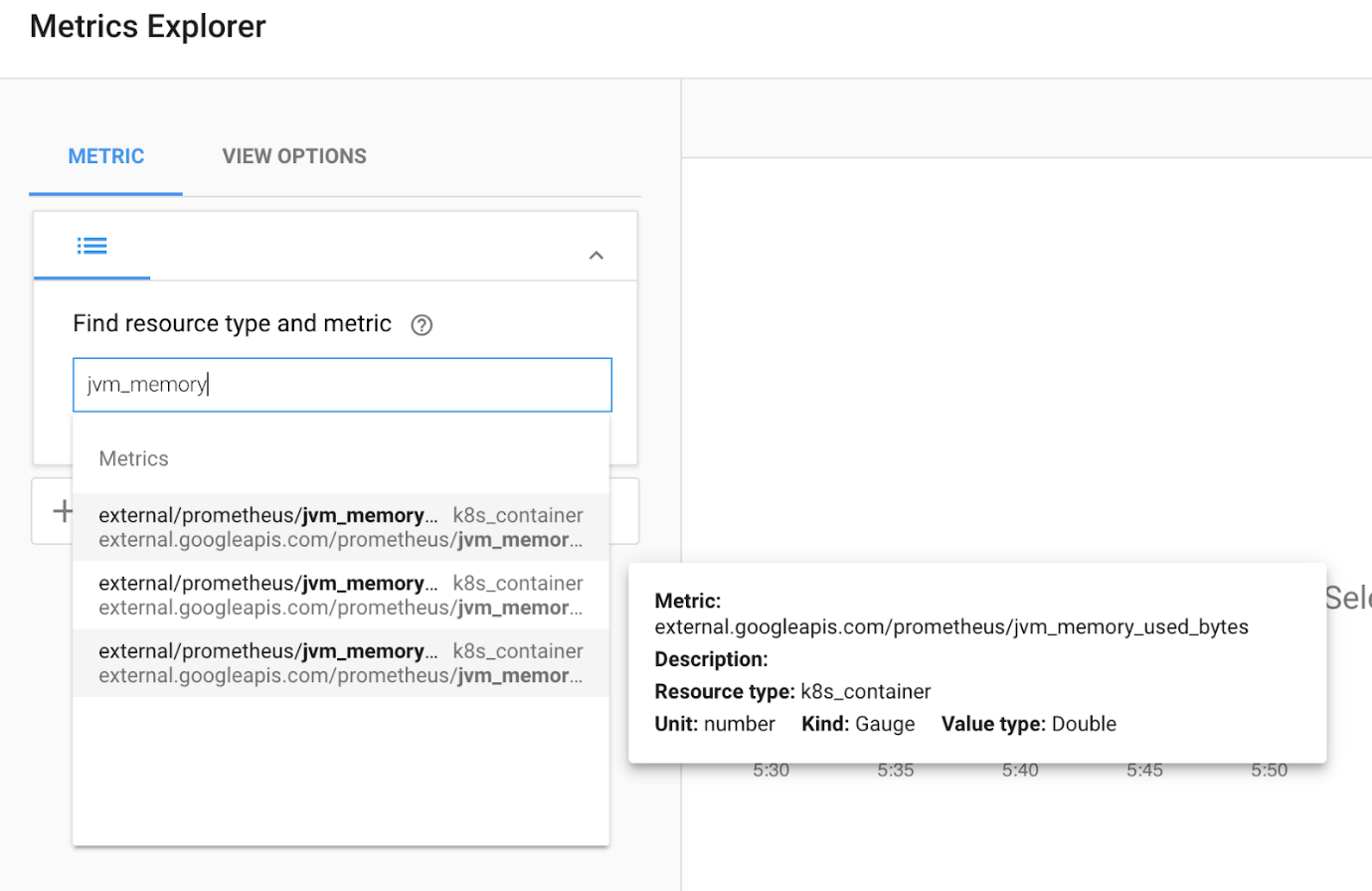


**Task 5. Explore the metrics**

1. In the Google Cloud Console, navigate to the **Operations** > **Monitoring**.

Refresh the page if the page is already open.

1. Click **Metrics Explorer**.
2. In the Metrics Explorer, search for jvm\_memory to find metrics collected by the Prometheus agent from the Spring Boot application.



1. Select jvm\_memory\_used\_bytes to plot the metrics. For **Resource Type**, select Kubernetes Container.

The JVM memory has multiple dimensions (for example, Heap versus Non-Heap and Eden Space versus Metaspace).

1. In **Filter**, filter by **area**, setting the value to **heap**, and in **Group By**, select **pod\_name**, and in **Aggregator**, select **sum**.

These options build a graph of current heap usage of the frontend application.

