

JAVAMS13 Working with Kubernetes Engine 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 Google Kubernetes Engine cluster with Kubernetes Engine Monitoring support. That means you can monitor the health of the GKE 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.

Kubernetes Engine Monitoring aggregates logs, events, and metrics from your GKE environment to help you understand your application's behavior in production. Prometheus is an optional monitoring tool often used with GKE. If you configure Kubernetes Engine 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 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 Monitoring to monitor internal performance metrics from your application while it is running on GKE.

Objectives

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

- Enable Monitoring for GKE
- Enable Prometheus monitoring in a GKE cluster
- Expose Prometheus metrics from inside a Spring Boot application
- Explore live application metrics using Monitoring


Setup and requirements


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.

[Open Google Console](#)

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. [Learn more.](#)

Username
google2727032_student@qwiklabs.n 

Password
k68CZXsxMZ 

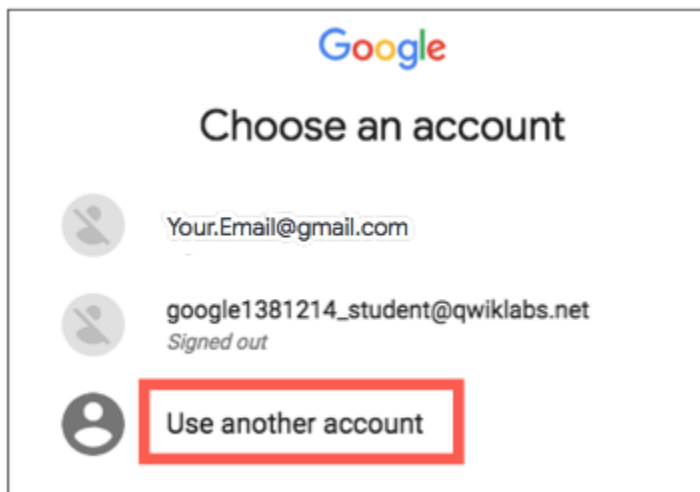
GCP Project ID
qwiklabs-gcp-4fbfecac8667e457 

[New to labs? View our introductory video!](#)

2. 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.

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

3. On the Choose an account page, click **Use Another Account**. The Sign in page opens.



4. Paste the username that you copied from the Connection Details panel. Then copy and paste the password.

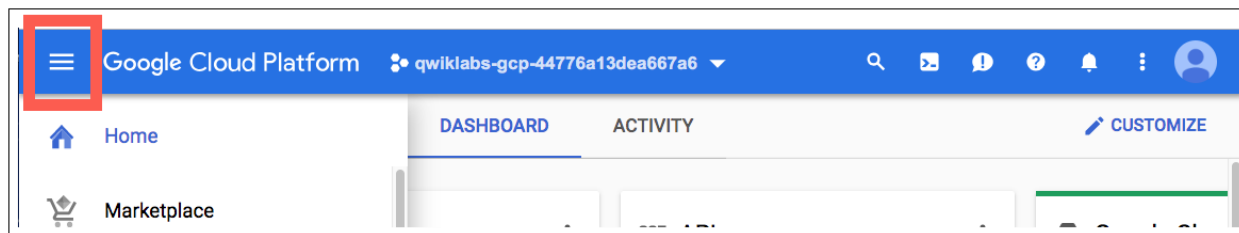
Note: You must use the credentials from the Connection Details panel. Do not use your Google Cloud Skills Boost credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

5. 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 Cloud console opens in this tab.

Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left.



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

Task 1. Fetch the application source files

In this task you clone the source repository files that are used throughout this lab.

1. To begin the lab, click the **Activate Cloud Shell** button at the top of the Google Cloud Console.
2. To activate the code editor, click the `Open Editor` button on the toolbar of the Cloud Shell window.

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.

3. In Cloud Shell, 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)')
```

Copied!

content_copy

4. Verify that the demo application files were created:

```
gsutil ls gs://$PROJECT_ID
```

Copied!

content_copy

Note: 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.

3. Copy the application folders to Cloud Shell:

```
gsutil -m cp -r gs://$PROJECT_ID/* ~/
```

Copied!

content_copy

4. Make the Maven wrapper scripts executable:

```
chmod +x ~/guestbook-frontend/mvnw
```

```
chmod +x ~/guestbook-service/mvnw
```

Copied!

content_copy

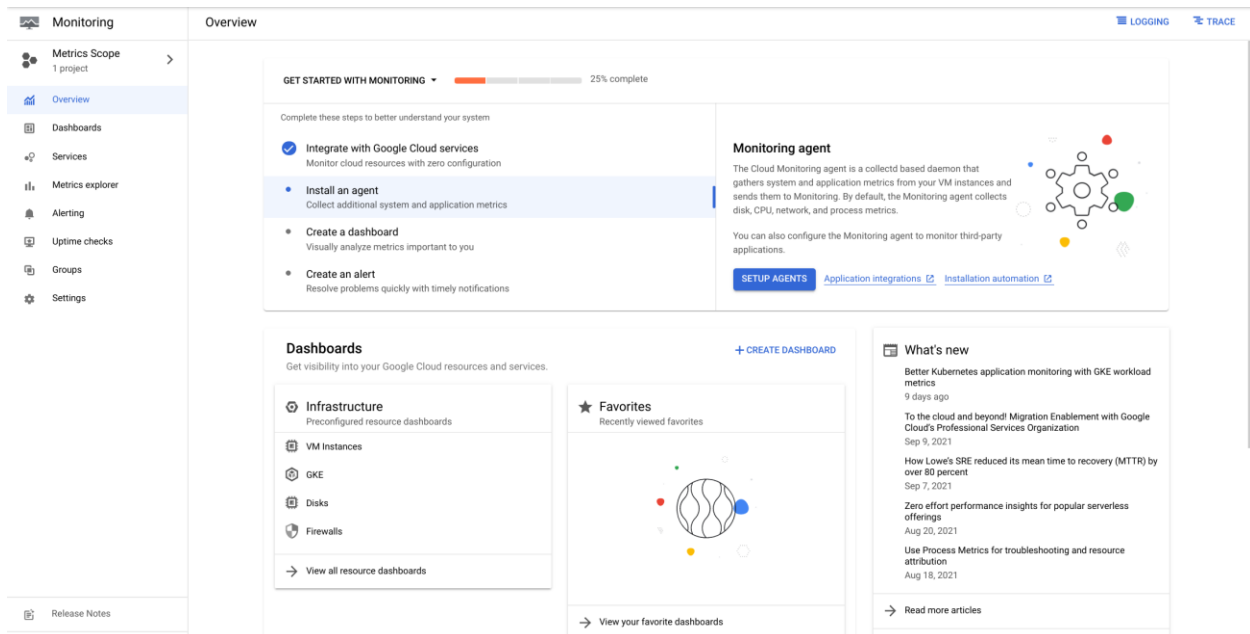
Task 2. Enable Monitoring and view the Kubernetes Engine Monitoring dashboard

Create a Monitoring workspace

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

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

When the Monitoring dashboard opens, your workspace is ready.



3. Click **Dashboards** and select **GKE** to view the Kubernetes Engine Monitoring dashboard.

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

Task 3. 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](#).

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>
```

Copied!

content_copy

3. In the Cloud Shell code editor, open `~/guestbook-frontend/src/main/resources/application.properties`.
4. 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=*
```

Copied!

content_copy

5. To send log entries to Cloud 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>

```

Copied!

content_copy

Task 4. Rebuild the containers

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

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

```
cd ~/guestbook-frontend
```

Copied!

content_copy

2. Build the frontend application container:

```
./mvnw clean compile jib:build
```

Copied!

content_copy

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

You update the GKE 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`.

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


```
- name: metrics
  containerPort: 9000
```

Copied!

content_copy

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

```
16  apiVersion: apps/v1
17  kind: Deployment
18  metadata:
19    labels:
20      app: guestbook-frontend
21      name: guestbook-frontend
22  spec:
23    replicas: 2
24    selector:
25      matchLabels:
26        app: guestbook-frontend
27    template:
28      metadata:
29        labels:
30          app: guestbook-frontend
31      spec:
32        volumes:
33          - name: credentials
34            secret:
35              secretName: guestbook-service-account
36        containers:
37          - name: guestbook-frontend
38            image: gcr.io/qwiklabs-gcp-02-d3683ac6304b/guestbook-frontend
39            volumeMounts:
40              - name: credentials
41                mountPath: "/etc/credentials"
42                readOnly: true
43            env:
44              - name: SPRING_CLOUD_CONFIG_ENABLED
45                value: "false"
46              - name: SPRING_CLOUD_GCP_CONFIG_ENABLED
47                value: "false"
48              - name: MESSAGES_ENDPOINT
49                value: http://guestbook-service:8080/guestbookMessages
50              - name: SPRING_PROFILES_ACTIVE
51                value: cloud
52              - name: GOOGLE_APPLICATION_CREDENTIALS
53                value: /etc/credentials/service-account.json
54            ports:
55              - name: http
56                containerPort: 8080
57              - name: metrics
58                containerPort: 9000
59
```

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

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

```
cd ~/guestbook-service
```

Copied!

content_copy

6. Build the guestbook backend service application container:

```
./mvnw clean compile jib:build
```

Copied!

content_copy

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.

7. 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 -
```

Copied!

content_copy

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

```
kubectl get pods -l app=guestbook-frontend
```

Copied!

content_copy

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

9. Establish a port forward to one of the Guestbook Frontend pods.

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
```

Copied!

content_copy

Task 5. Install Prometheus and Sidecar

Kubernetes Engine Monitoring can [monitor Prometheus metrics](#) from the GKE cluster. Install Prometheus support to the cluster.

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

```
export PROMETHEUS_VERSION=v0.58.0
gcloud container clusters get-credentials guestbook-cluster --zone=us-central1-a
kubectl apply -f https://raw.githubusercontent.com/coreos/prometheus-operator/${PROMETHEUS_VERSION}/bundle.yaml --force-conflicts=true --server-side
```

Copied!

content_copy

2. 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
```

Copied!

content_copy


Note: The prometheus.yaml file has an additional Prometheus Sidecar that's designed to export the scraped Prometheus metrics to Cloud Operations.

3. 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
```

Copied!

content_copy

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

Note: The guestbook is not running locally, so you won't see the guestbook page.

5. 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://9090-93a373eb-dd32-458e-b262-3f24c944f746.q1-us-west1-jrts.cloudshell.dev/graph`.

6. In the Prometheus console, select **Status > Targets**.

7. Observe that there are 2 targets (2 pods) being scrapped for metrics.

Task 6. Explore the metrics

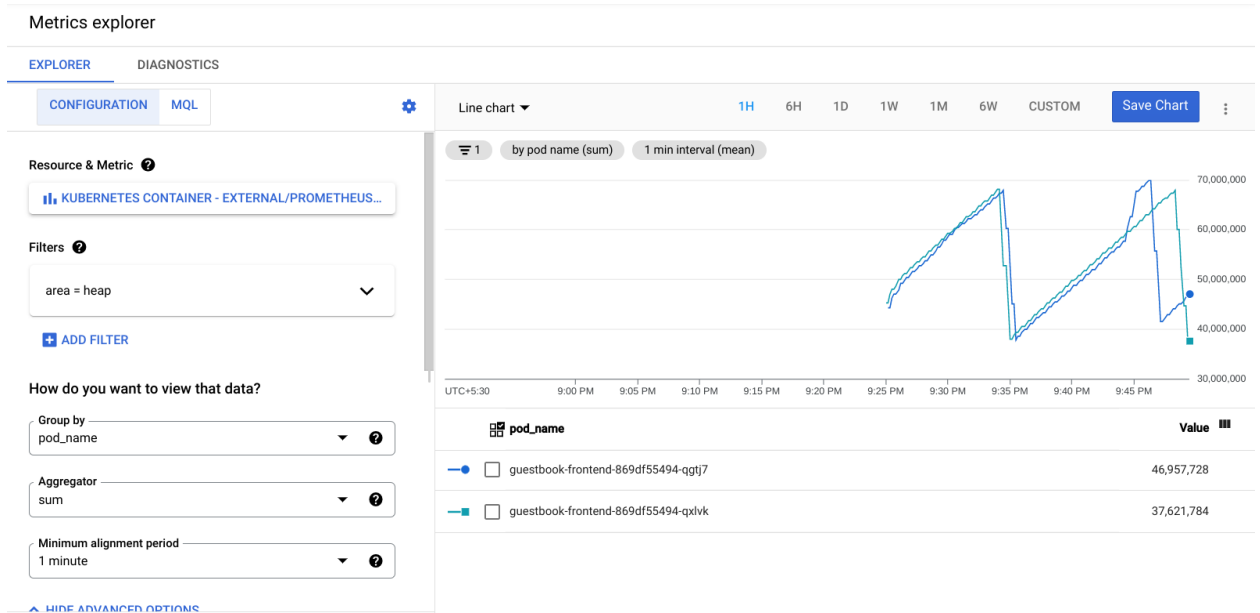
1. In the Google Cloud Console, navigate to the **Operations > Monitoring**.
2. Click **Metrics Explorer**.
3. In the Metrics Explorer, for **Resource & Metric**, turn off **Show only active resources & metrics**, and then select **Kubernetes Container > Prometheus** and select **jvm_memory_used_bytes`**.
4. Click **Apply**.

Note: It may take a few minutes for the Prometheus metrics to show up in the Metrics Explorer.

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

5. In **Filter**, click **+Add Filter**.
6. For the label, select **area**, set the value to **heap**, and then click **Done**.
7. In **Group By**, select **pod_name** and click **OK**. In **Aggregator**, select **sum**.

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



Task 7. Review

In this lab you enabled Monitoring for GKE. You also enabled Prometheus monitoring in a GKE cluster, and exposed Prometheus metrics from inside a Spring Boot application. Finally, you explored live application metrics using Monitoring.