Building Scalable Java Microservices with Spring Boot and Spring Cloud

AGENDA

- Concept Study
- Introduction to Google Cloud Services and Spring boot
- The Demo Java Microservices Application
- Conclusion

Concept Study

The Spring Framework is an open-source application framework for Java. Spring Boot is a solution that makes it easier to build Spring Framework applications. Spring Boot adopts the "convention over configuration" paradigm, which means that developers should only have to specify unusual aspects of the applications they build. Spring Boot autoconfigures as much as possible using sensible defaults. Spring Cloud makes Google Cloud services available to Spring Framework applications.

We will learn how to integrate Google Cloud services with our Java applications

Introduction to Google Cloud Services and Spring boot

We'll introduce the GCP technologies that will be covered for this project, and describe how Spring Boot simplifies a task of using external services, such as GCP services in your applications. We will learn about the demo Java application that is used for this project, and the GCP services that you'll integrate with that application. I'll also introduce Spring Boot, and explain how it is used in the project.

❑ Google Cloud services overview

Google Cloud Platform providers a range of cloud services that are options for solution architects and developers such as :

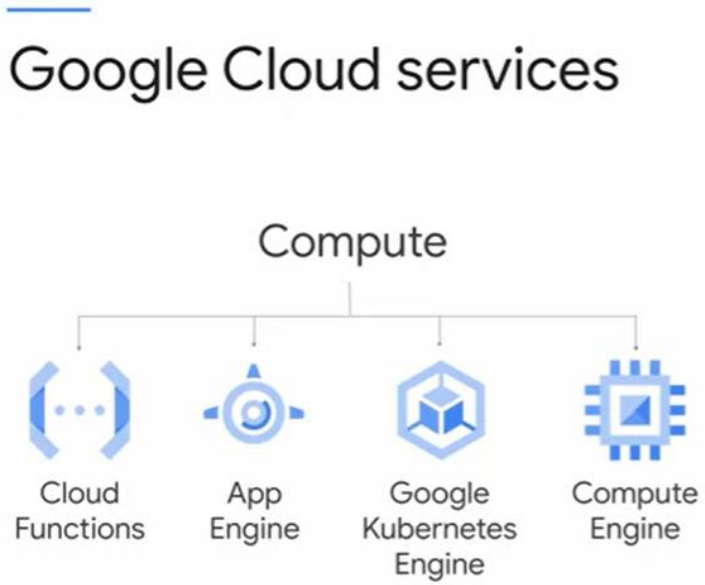
- **Compute and Hosting services**
- **Storage services**
- **Big Data and Machine Learning services**

Let’s jump more in depth into each service

Introduction to Google Cloud Services and Spring boot

❑ Compute and Hosting services

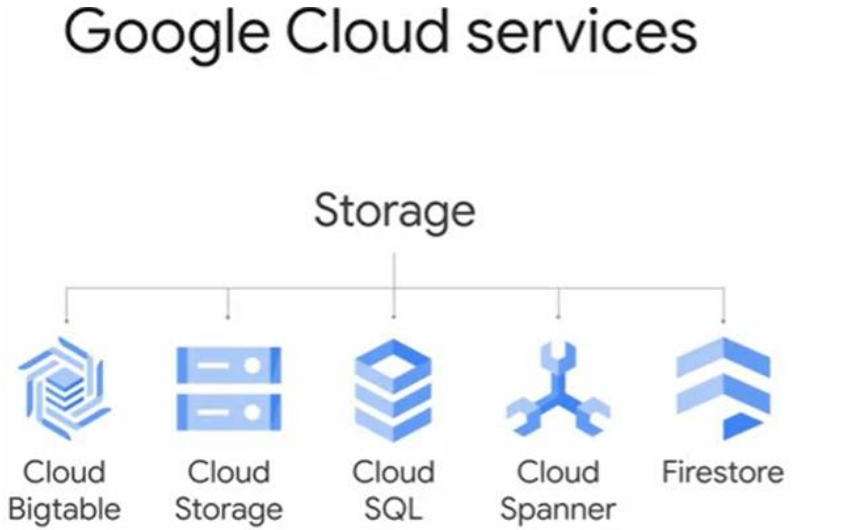
GCP provides a range of compute and hosting services including: Cloud functions that provide a completely serverless execution environment or functions as a service; App Engine that provides a fully managed platform as a service framework; Kubernetes engine that provides a managed containers as a service environment for containerized applications, and Compute Engine which is Google's infrastructure as a service solution that providers maximum flexibility for users and organizations who choose to manage their solutions themselves. Below is a picture of these services.



Introduction to Google Cloud Services and Spring boot

❑ Storage services

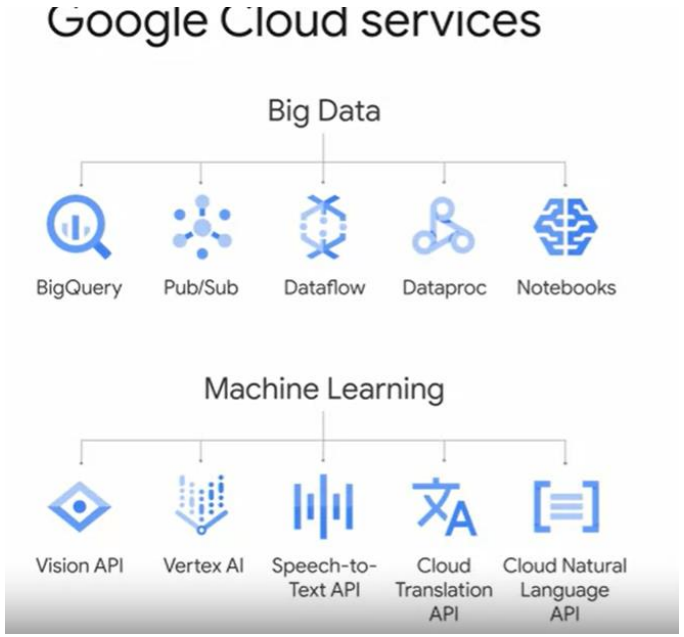
GCP also provides a range of storage services. These services can deliver cost and performance-optimized solutions for structured, unstructured, transactional, and relational data. Cloud storage is used for persistent storage of unstructured file data and update structured relational database content in Cloud SQL. Cloud SQL instance and Cloud Spanner instance are used to meet the needs for high levels of transactional performance. With Cloud Firestore, you can automatically synchronize your app data between devices. Cloud Bigtable is a fully managed, scalable NoSQL database service for large analytical and operational workloads with up to 99.999% availability. Below is a picture of these services.



Introduction to Google Cloud Services and Spring boot

❑ Big Data and Machine Learning services

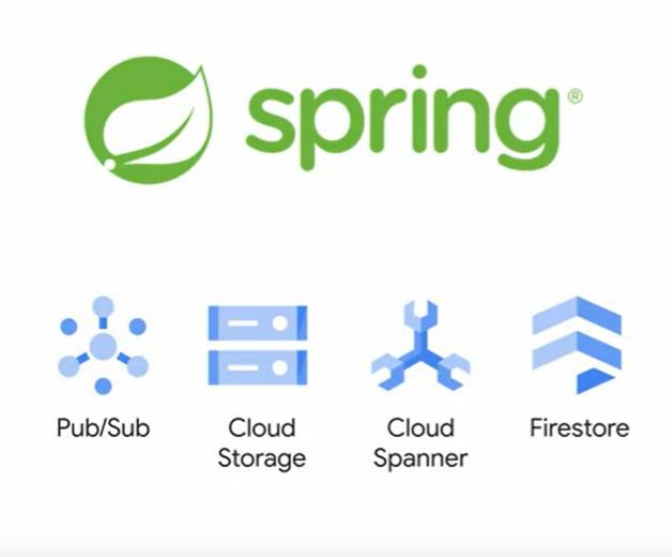
GCP Big Data and machine learning services allow you to turn data into actionable insights with a comprehensive set of data analytics and machine learning services. The fully managed serverless approach of GCP removes operational overhead, handling your Big Data analytic solutions performance, scalability, availability, security, and compliance needs automatically. This allows you to focus on analysis instead of managing service. Below is a picture of these services.



Introduction to Google Cloud Services and Spring boot

❑ Spring Framework introduction

The Spring framework is an open-source application framework and inversion of control container developed by Pivotal for the Java platform. Spring Boot is a Spring framework solution designed to simplify the bootstrapping and development of standalone production spring-based application. Spring Boot provides pre-configured starters that provide automatic quick-start configurations for the Spring platform with third-party libraries. Spring Cloud GCP makes it easy for Spring users to run their applications on GCP by providing a wide range of Spring Boot starters for various GCP services, and direct support for Cloud Pub/Sub, Cloud Storage, Cloud Spanner, and Cloud Datastore



The Demo Java Microservices Application

❑ The demo application architecture

The demo application is composed in two separate microservices-style compliment applications: the Guestbook frontend application, and the Guestbook backend service . They are written in Java with Spring Boot. The Guestbook application delivers basic CRUD functionality, although only the create and retrieve functions are used.

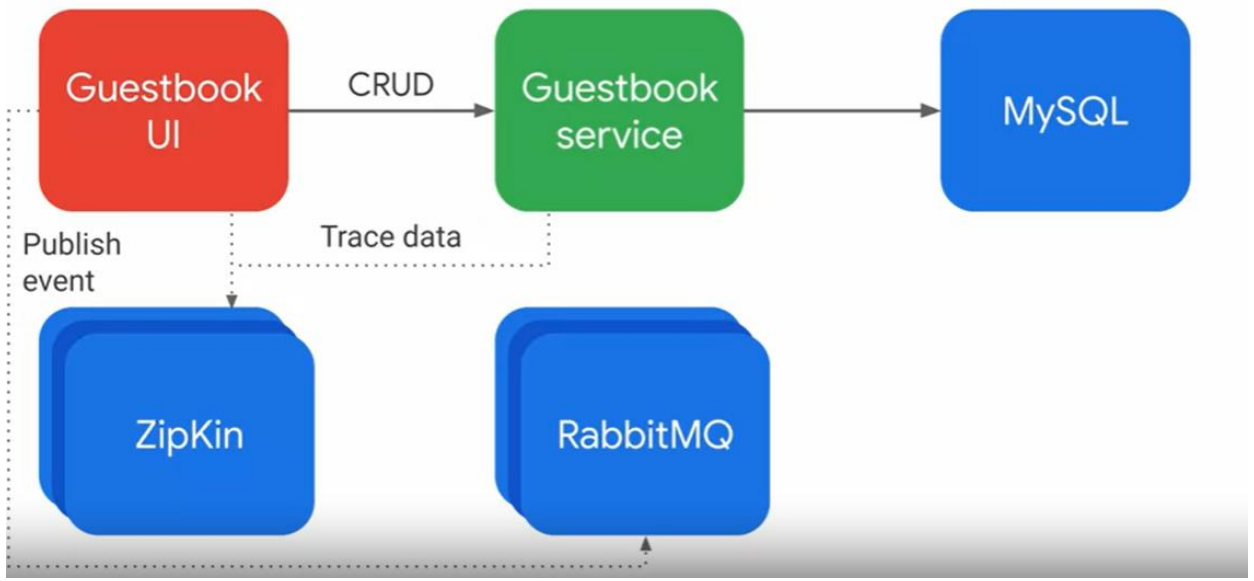
The demo application follows a classic three-tier web architecture illustrated by the picture below.

The Demo Java Microservices Application

❑ The demo application architecture

Although the demo application is quite simple, it allows us to demonstrate how to implement distributed tracing using a system like ZipKin. ZipKin is able to provide detailed timing data that can be used to troubleshoot latency issues with a microservices-style application like this. The communication between services is a critical competent of a microservices framework. And a message queuing solutions, like RabbitMQ, allow services to send or receive messages or requests asynchronously instead of having to carry out research-intensive processing at the same time

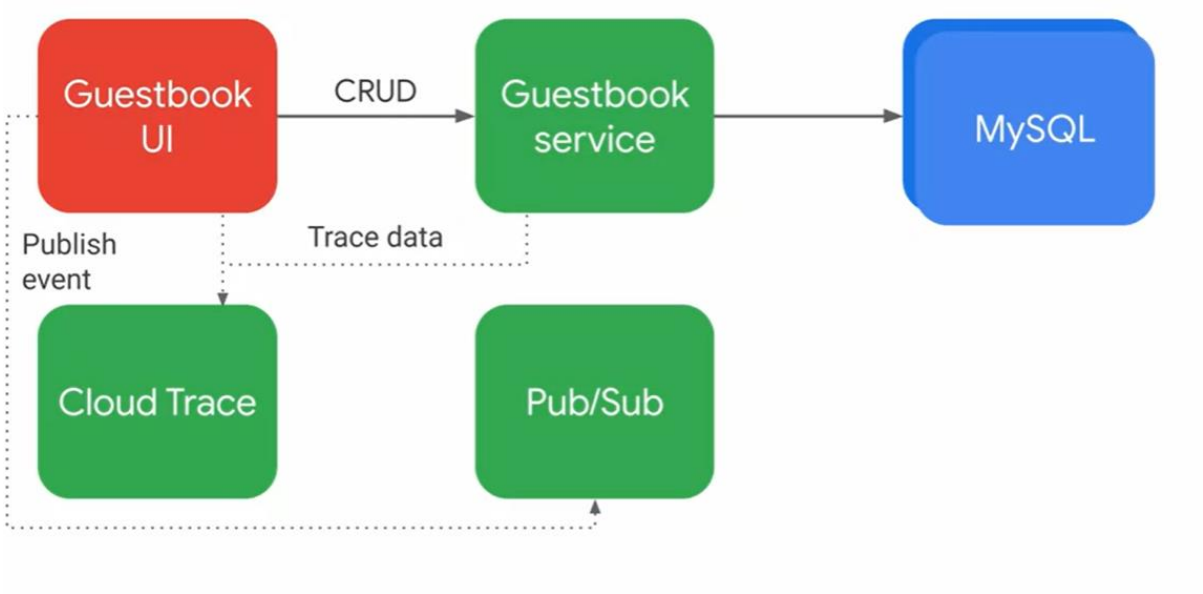
This architecture is more detailed by the picture below.



The Demo Java Microservices Application

❑ The demo application architecture

If you want to implement a solution that includes some or all of these components, that additional cost and operational overhead must be allowed for. So, adopting a Cloud-native approach through the use of GCP services for these capabilities, instead of building and managing your own, provides instant access to the capability without having to deploy any additional infrastructure yourself. This also avoids the overhead and complexity of managing the lifecycle of trace, messaging queue, database, and runtime configuration service servers. This architecture is more detailed by the picture below.

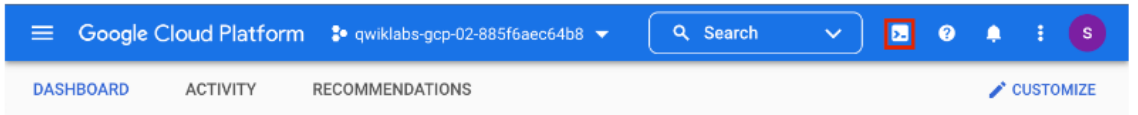


The Demo Java Microservices Application

❑ Bootstrapping the application frontend and backend

For this part, we will take a demo microservices Java application built with the Spring framework and modify it to use an external database server. We will adopt some of the best practices for tracing, configuration management, and integration with other services using integration patterns. We will proceed as follow.

1. In Cloud console, on the top right toolbar, click the Open Cloud Shell button.



2. Click **Continue**.

The Demo Java Microservices Application

❑ Bootstrapping the application frontend and backend

Task 1. Bootstrap the application

In this task, you clone the source repository for the demo application that is used throughout these labs. The demo application has two parts:

- A frontend application (`guestbook-frontend`) that manages the user interface presented in a web browser
- A backend service application (`guestbook-service`) that processes the data and manages the messaging and database interfaces

Clone the demo application

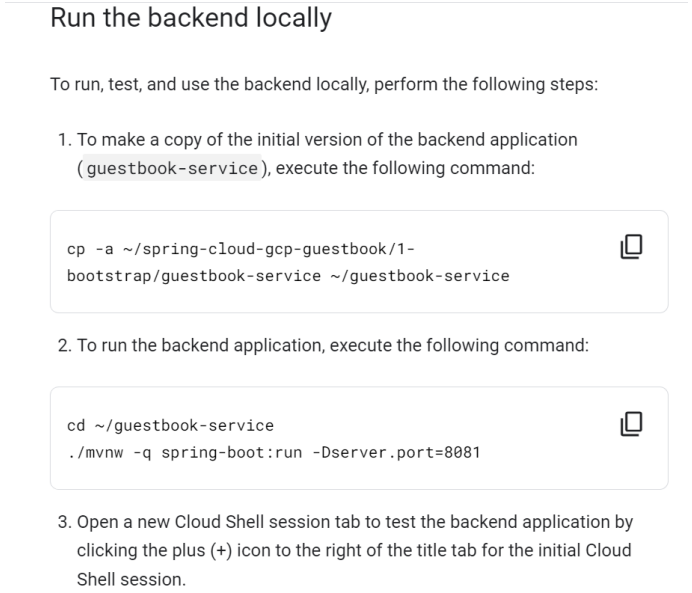
1. In Cloud Shell, clone the demo application by executing the following command:

```
cd ~/
git clone --depth=1
https://github.com/GoogleCloudPlatform/training-data-analyst
ln -s ~/training-data-analyst/courses/java-microservices/spring-cloud-gcp-guestbook ~/spring-cloud-gcp-guestbook
```

cd ~/
git clone --depth=1
https://github.com/GoogleCloudPlatform/training-data-analyst
ln -s ~/training-data-analyst/courses/java-microservices/spring-cloud-gcp-guestbook ~/spring-cloud-gcp-guestbook

The Demo Java Microservices Application

- Bootstrapping the application frontend and backend



Run the backend locally

To run, test, and use the backend locally, perform the following steps:

1. To make a copy of the initial version of the backend application (`guestbook-service`), execute the following command:

```
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-service ~/guestbook-service
```

2. To run the backend application, execute the following command:

```
cd ~/guestbook-service
./mvnw -q spring-boot:run -Dserver.port=8081
```

3. Open a new Cloud Shell session tab to test the backend application by clicking the plus (+) icon to the right of the title tab for the initial Cloud Shell session.

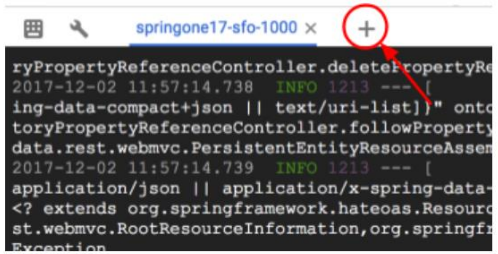
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-service ~/guestbook-service

cd ~/guestbook-service
./mvnw -q spring-boot:run -Dserver.port=8081

The Demo Java Microservices Application

❑ Bootstrapping the application frontend and backend

This action opens a second Cloud Shell console to the same virtual machine.



4. While the backend application (`guestbook-service`) is still running, test the service by executing the following command in the second Cloud Shell tab:

```
curl http://localhost:8081/guestbookMessages
```

5. Post a new message.

```
curl -XPOST -H "content-type: application/json" \
  -d '{"name": "Ray", "message": "Hello"}' \
  http://localhost:8081/guestbookMessages
```

6. List all the messages.

```
curl http://localhost:8081/guestbookMessages
```

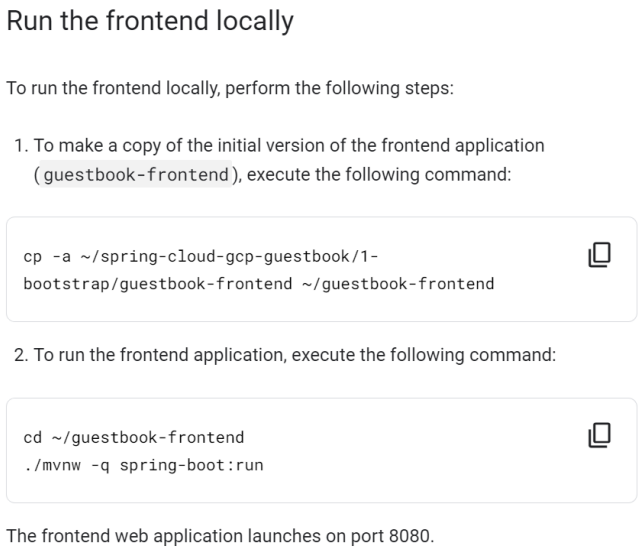
curl http://localhost:8081/guestbookMessages

curl -XPOST -H "content-type: application/json" \
  -d '{"name": "Ray", "message": "Hello"}' \
  http://localhost:8081/guestbookMessages

curl http://localhost:8081/guestbookMessages

The Demo Java Microservices Application

❑ Bootstrapping the application frontend and backend



Run the frontend locally

To run the frontend locally, perform the following steps:

1. To make a copy of the initial version of the frontend application (`guestbook-frontend`), execute the following command:

```
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-frontend ~/guestbook-frontend
```

2. To run the frontend application, execute the following command:

```
cd ~/guestbook-frontend
./mvnw -q spring-boot:run
```

The frontend web application launches on port 8080.

cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-frontend ~/guestbook-frontend

cd ~/guestbook-frontend
./mvnw -q spring-boot:run

The Demo Java Microservices Application

❑ Bootstrapping the application frontend and backend

Task 2. Test the guestbook application

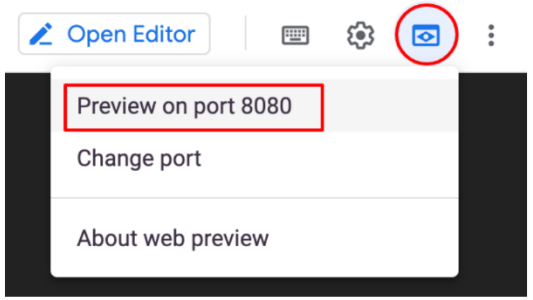
In this task, you test the demo application. The demo application is a simple Java application composed of a microservices backend, and a frontend consuming it. You extend this simple application in later labs to leverage various Google Cloud services. You eventually deploy it to the cloud, using both App Engine and Google Kubernetes Engine.

Access the frontend application through the Cloud Shell web preview

To access and use the frontend application through the web preview, perform the following steps:

1. Click **Web Preview** (🖵).

2. Select **Preview on port 8080** to access the application on port 8080.



A new browser tab displays the connection to the frontend application.

3. For **Your name**, type your name.

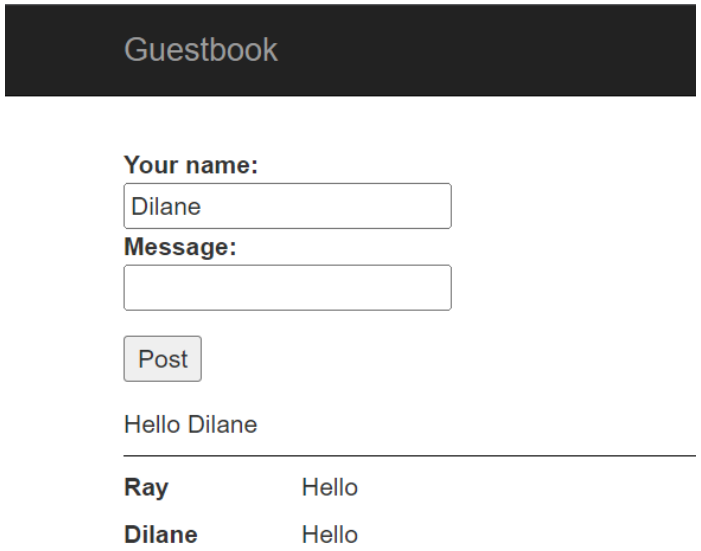
4. For **Message**, type a message.

5. Click **Post** to continue.

The messages are listed below the message input section.

The Demo Java Microservices Application

- Bootstrapping the application frontend and backend



Use Cloud Shell to test the backend service

To use Cloud Shell to test the backend service, perform the following steps:

1. Open a third Cloud Shell tab.
2. In the new shell tab, list all the messages that you added through a call to the backend `guestbook-service` API by executing the following command:

```
curl -s http://localhost:8081/guestbookMessages
```

3. To use `jq` to parse the JSON return text, execute the following command:

For example, the following command prints out only the messages:

```
curl -s http://localhost:8081/guestbookMessages \
  | jq -r '._embedded.guestbookMessages[] | {name:
.name, message: .message}'
```

```
curl -s http://localhost:8081/guestbookMessages
```

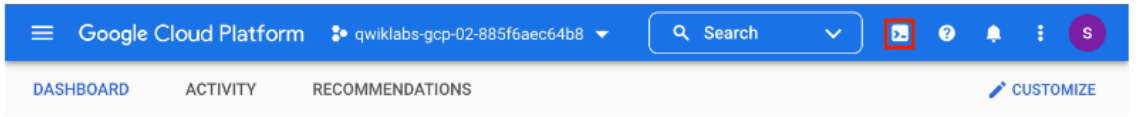
```
curl -s http://localhost:8081/guestbookMessages \
  | jq -r '._embedded.guestbookMessages[] | {name: .name,
message: .message}'
```

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

For this part, we will take the demo application and extend it to make use of Cloud SQL for database storage, using the Java Persistence API, or JPA, and Hibernate with Spring data framework

1. In Cloud console, on the top right toolbar, click the Open Cloud Shell button.



2. Click **Continue**.

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Task 1. Fetch the application source files

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

1. To clone the project repository, execute the following commands:

```
git clone --depth=1
https://github.com/GoogleCloudPlatform/training-data-analyst
ln -s ~/training-data-analyst/courses/java-microservices/spring-cloud-gcp-guestbook ~/spring-cloud-gcp-guestbook
```

2. Copy the relevant folders to your home directory:

```
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-service ~/guestbook-service
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-frontend ~/guestbook-frontend
```

git clone --depth=1 https://github.com/GoogleCloudPlatform/training-data-analyst
ln -s ~/training-data-analyst/courses/java-microservices/spring-cloud-gcp-guestbook ~/spring-cloud-gcp-guestbook

cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-service ~/guestbook-service
cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-frontend ~/guestbook-frontend

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Task 2. Create a Cloud SQL instance, database, and table

In this task, you provision a new Cloud SQL instance, create a database on that instance, and then create a database table with a schema that can be used by the demo application to store messages.

Enable Cloud SQL Administration API

You enable Cloud SQL Administration API and verify that Cloud SQL is preprovisioned.

1. In Cloud Shell, enable the Cloud SQL Administration API:

```
gcloud services enable sqladmin.googleapis.com
```

2. Confirm that Cloud SQL Administration API is enabled:

```
gcloud services list | grep sqladmin
```

3. List the Cloud SQL instances:

```
gcloud sql instances list
```

No instances are listed yet.

gcloud services enable sqladmin.googleapis.com

gcloud services list | grep sqladmin

gcloud sql instances list

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Create a Cloud SQL instance

You create a new Cloud SQL instance.

1. Provision a new Cloud SQL instance:

```
gcloud sql instances create guestbook --region=us-
central1
```

Provisioning the Cloud SQL instance will take a couple of minutes to complete.

The output should look similar to this:

```
Creating Cloud SQL instance...done.
Created [...].
NAME       DATABASE_VERSION REGION       TIER              ADDRE
guestbook  MYSQL_5_6        us-central1  db-n1-standard-1  92.3.
```

Create a database in the Cloud SQL instance

You create a database to be used by the demo application.

1. Create a `messages` database in the MySQL instance:

```
gcloud sql databases create messages --instance
guestbook
```

gcloud sql instances create guestbook --region=us-central1

gcloud sql databases create messages --instance guestbook

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Connect to Cloud SQL and create the schema

By default, Cloud SQL is not accessible through public IP addresses. You can connect to Cloud SQL in the following ways:

- Use a local Cloud SQL proxy.
- Use `gcloud` to connect through a CLI client.
- From the Java application, use the MySQL JDBC driver with an SSL socket factory for secured connection.

You create the database schema to be used by the demo application to store messages.

1. Use `gcloud` CLI to connect to the database:

```
gcloud sql connect guestbook
```

This command temporarily allowlists the IP address for the connection.

2. Press ENTER at the following prompt to leave the password empty for this lab.

```
Allowlisting your IP for incoming connection for 5
minutes...done.
Connecting to database with SQL user [root].Enter password:
```

The root password is empty by default.

The prompt changes to mysql> to indicate that you are now working in the MySQL command-line environment.

gcloud sql connect guestbook

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

3. List the databases:

```
show databases;
```

This command lists all of the databases on the Cloud SQL instance, which should include the messages database that you configured in previous steps.

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| messages           |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.04 sec)
```

4. Switch to the `messages` database:

```
use messages;
```

5. Create the table:

```
CREATE TABLE guestbook_message (
  id BIGINT NOT NULL AUTO_INCREMENT,
  name CHAR(128) NOT NULL,
  message CHAR(255),
  image_uri CHAR(255),
  PRIMARY KEY (id)
);
```

6. Exit the MySQL management utility by executing the following command:

```
exit
```

```
show databases;
use messages;
CREATE TABLE guestbook_message (
  id BIGINT NOT NULL AUTO_INCREMENT,
  name CHAR(128) NOT NULL,
  message CHAR(255),
  image_uri CHAR(255),
  PRIMARY KEY (id)
);
exit
```

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Task 3. Use Spring to add Cloud SQL support to your application

In this task, you add the Spring Cloud GCP Cloud SQL starter to your project so that you can use Spring to connect to your Cloud SQL database.

Add the Spring Cloud GCP Cloud SQL starter

From a Java application, you can integrate with a Cloud SQL instance by using the standard method, where you use the JDBC driver. However, configuring the JDBC driver for use with Cloud SQL can be more complicated than connecting to a standard MySQL server because of the additional security that Google Cloud puts in place. Using the Spring Cloud GCP Cloud SQL starter simplifies this task.

The Spring Cloud GCP project provides configurations that you can use to automatically configure your Spring Boot applications to consume Google Cloud services, including Cloud SQL.

You update the guestbook service's `pom.xml` file to import the Spring Cloud GCP BOM and also the Spring Cloud GCP Cloud SQL starter. This process involves adding the milestone repository to use the latest Spring release candidates.

1. Edit `guestbook-service/pom.xml` in the Cloud Shell code editor or in the shell editor of your choice.

2. Insert an additional dependency for `spring-cloud-gcp-starter-sql-mysql` just before the `</dependencies>` closing tag:

```
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-starter-sql-
mysql</artifactId>
    </dependency>
```

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Disable Cloud SQL in the default profile

For local testing, you can continue to use a local database or an embedded database. The demo application is initially configured to use an embedded HSQL database.

To continue to use the demo application for local runs, you disable the Cloud SQL starter in the default application profile by updating the `application.properties` file.

1. In the Cloud Shell code editor, open `guestbook-service/src/main/resources/application.properties`.
2. Add the following property setting:

```
spring.cloud.gcp.sql.enabled=false
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-sql-mysql</artifactId>
</dependency>
```

spring.cloud.gcp.sql.enabled=false

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Task 4. Configure an application profile to use Cloud SQL

In this task, you create an application profile that contains the properties that are required by the Spring Boot Cloud SQL starter to connect to your Cloud SQL database.

Configure a cloud profile

When deploying the demo application into the cloud, you want to use the production-managed Cloud SQL instance.

You create an application profile called `cloud` profile. The `cloud` profile leaves the Cloud SQL starter that is defined in the Spring configuration profile enabled. And it includes properties used by the Cloud SQL starter to provide the connection details for your Cloud SQL instance and database.

1. In Cloud Shell, find the instance connection name:

```
gcloud sql instances describe guestbook --
format='value(connectionName)'
```

This command format filters out the `connectionName` property from the description of the guestbook Cloud SQL object. The entire string that is returned is the instance's connection name. The string looks like the following example:

```
qwiklabs-gcp-4d0ab38f9ff2cc4c:us-central1:guestbook
```

2. In the Cloud Shell code editor **create** an `application-cloud.properties` file in the `guestbook-service/src/main/resources` directory.

gcloud sql instances describe guestbook --
format='value(connectionName)'

The Demo Java Microservices Application

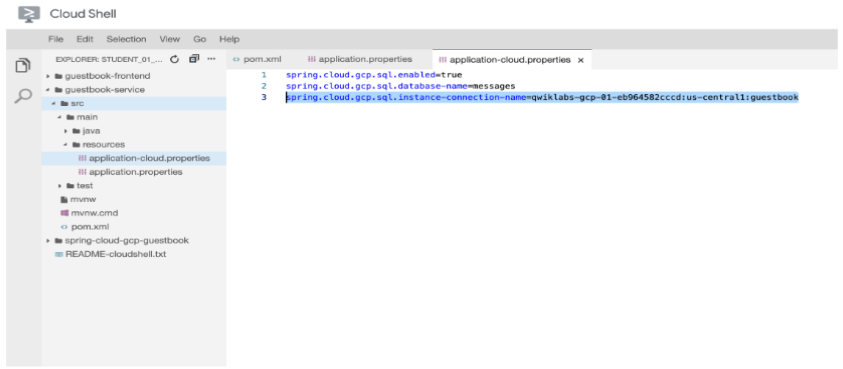
❑ Configuring and connecting to Cloud SQL

3. In the Cloud Shell code editor, open `guestbook-service/src/main/resources/application-cloud.properties` and add the following properties:

```
spring.cloud.gcp.sql.enabled=true
spring.cloud.gcp.sql.database-name=messages
spring.cloud.gcp.sql.instance-connection-
name=YOUR_INSTANCE_CONNECTION_NAME
```

4. Replace the `YOUR_INSTANCE_CONNECTION_NAME` placeholder with the full connection name string returned in step 1 in this task.

If you worked in the Cloud Shell code editor, your screen looks like the following screenshot:



spring.cloud.gcp.sql.enabled=true
spring.cloud.gcp.sql.database-name=messages
spring.cloud.gcp.sql.instance-connection-
name=YOUR_INSTANCE_CONNECTION_NAME

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

Configure the connection pool

You use the `spring.datasource.*` configuration properties to configure the JDBC connection pool, as you do with other Spring Boot applications.

1. Add the following property to `guestbook-service/src/main/resources/application-cloud.properties` that should still be open in the Cloud Shell code editor to specify the connection pool size:

```
spring.datasource.hikari.maximum-pool-size=5
```

Test the backend service running on Cloud SQL

You relaunch the backend service for the demo application in Cloud Shell, using the new cloud profile that configures the service to use Cloud SQL instead of the embedded HSQL database.

1. In Cloud Shell, change to the `guestbook-service` directory:

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

2. Run a test with the default profile and make sure there are no failures:

```
./mvnw test
```

spring.datasource.hikari.maximum-pool-size=5

cd ~/guestbook-service

./mvnw test

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

3. Start the Guestbook Service with the cloud profile:

```
./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud"
```

4. During the application startup, validate that you see CloudSQL related connection logs.

The output should look similar to this:

```
... First Cloud SQL connection, generating RSA key pair.
... Obtaining ephemeral certificate for Cloud SQL instance [spri
... Connecting to Cloud SQL instance [...:us-central1:guestbook]
... Connecting to Cloud SQL instance [...:us-central1:guestbook]
... Connecting to Cloud SQL instance [...:us-central1:guestbook]
...
```

5. In a new Cloud Shell tab, make a few calls using `curl`:

```
curl -XPOST -H "content-type: application/json" \
  -d '{"name": "Ray", "message": "Hello CloudSQL"}' \
  http://localhost:8081/guestbookMessages
```

6. You can also list all the messages:

```
curl http://localhost:8081/guestbookMessages
```

7. Use the Cloud SQL client to validate that message records have been added to the database:

```
gcloud sql connect guestbook
```

8. Press ENTER at the Enter password prompt.

./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud“

curl -XPOST -H "content-type: application/json" \
  -d '{"name": "Ray", "message": "Hello CloudSQL"}' \
  http://localhost:8081/guestbookMessages

curl http://localhost:8081/guestbookMessages

gcloud sql connect guestbook

The Demo Java Microservices Application

❑ Configuring and connecting to Cloud SQL

9. Query the `guestbook_message` table in the messages database:

```
use messages;
select * from guestbook_message;
```

The `guestbook_messages` table now contains a record of the test message that you sent using `curl` in a previous step.

The output should look similar to this:

```
+----+------+----------------+-----------+
| id | name | message        | image_uri |
+----+------+----------------+-----------+
|  1 | Ray  | Hello Cloud SQL | NULL      |
+----+------+----------------+-----------+
1 row in set (0.04 sec)
```

9. Close the Cloud SQL interactive client by executing the following command:

```
exit
```

use messages;
select * from guestbook_message;

exit

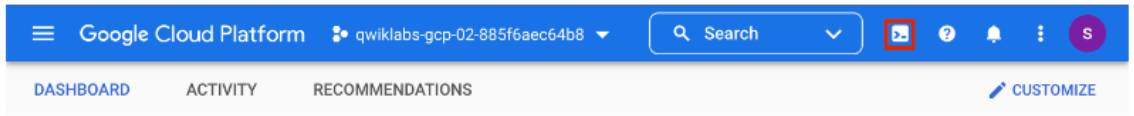
The Demo Java Microservices Application

❑ Working with Cloud Trace

Cloud Trace is a distributed tracing system that collects latency data from your applications and displays it in the Google Cloud console.

For this part, we will implement distributed tracing by using Spring Cloud GCP, Spring Cloud Sleuth, and Cloud Trace. Spring Cloud GCP provides a starter that can interoperate with Spring Cloud Sleuth, but it forwards the trace request to Cloud Trace instead.

1. In Cloud console, on the top right toolbar, click the Open Cloud Shell button.



2. Click **Continue**.

The Demo Java Microservices Application

❑ Working with Cloud Trace

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.

You can switch between Cloud Shell and the code editor by using `Open Editor` and `Open Terminal` icon as required.

6. Make the Maven wrapper scripts executable:

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

Now you're ready to go!

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)')
```

4. Verify that the demo application files were created:

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

5. Copy the application folders to Cloud Shell:

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

export PROJECT_ID=$(gcloud config list --format 'value(core.project)')

gsutil ls gs://$PROJECT_ID

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

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

The Demo Java Microservices Application

❑ Working with Cloud Trace

Task 2. Enable Cloud Trace API

In this task, you enable the Cloud Trace API.

- In Cloud Shell, enter the following command:

```
gcloud services enable cloudtrace.googleapis.com
```

gcloud services enable cloudtrace.googleapis.com

The Demo Java Microservices Application

❑ Working with Cloud Trace

Task 3. Add the Spring Cloud GCP Trace starter

In this task, you add the Spring Cloud GCP Trace starter to the `pom.xml` files for the guestbook service and the guestbook frontend applications.

1. Click on the `Open Editor` icon and then, open `~/guestbook-service/pom.xml`.

2. Insert the following new dependency at the end of the dependencies section, just before the closing `</dependencies>` tag:

```
        <dependency>

<groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gcp-starter-
trace</artifactId>
        </dependency>
```

3. In the Cloud Shell code editor, open `~/guestbook-frontend/pom.xml`.

4. Insert the following new dependency at the end of the dependencies section, just before the closing `</dependencies>` tag:

```
        <dependency>

<groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gcp-starter-
trace</artifactId>
        </dependency>
```

Copy

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-gcp-starter-trace</artifactId>
</dependency>
```

The Demo Java Microservices Application

❑ Working with Cloud Trace

Task 4. Configure applications

In this task, you disable trace completely for the default profile and configure trace sampling for all requests in the `cloud` profile.

You get full trace capability simply by including the starters. However, only a small percentage of all requests have their traces sampled and stored by default.

Disable trace for testing purposes

For testing purposes, you disable trace in the `application.properties` files used for the local profile.

1. In the Cloud Shell code editor, open the application properties:

```
guestbook-
service/src/main/resources/application.properties
```

2. Add the following property to disable tracing in the guestbook service:

```
spring.cloud.gcp.trace.enabled=false
```

3. In the Cloud Shell code editor, open the following:

```
guestbook-
frontend/src/main/resources/application.properties`.
```

4. Add the following property to disable tracing in the guestbook frontend application:

```
spring.cloud.gcp.trace.enabled=false
```

spring.cloud.gcp.trace.enabled=false

The Demo Java Microservices Application

❑ Working with Cloud Trace

Enable trace sampling for the cloud profile for the guestbook backend

For the `cloud` profile for the guestbook backend, you enable trace sampling for all of the requests in the `application.properties` file used for the `cloud` profile.

1. In the Cloud Shell code editor, open the guestbook service `cloud` profile:

```
guestbook-service/src/main/resources/application-
cloud.properties`.
```

2. Add the following properties to enable the tracing detail needed in the guestbook service:

```
spring.cloud.gcp.trace.enabled=true
spring.sleuth.sampler.probability=1.0
spring.sleuth.scheduled.enabled=false
```

Enable trace sampling for the cloud profile for the guestbook frontend

For the `cloud` profile for the frontend application, you enable trace sampling for all of the requests in the `application.properties` file used for the `cloud` profile.

1. In the Cloud Shell code editor, **create** a properties file for the guestbook frontend application `cloud` profile:

```
guestbook-frontend/src/main/resources/application-
cloud.properties
```

2. Add the following properties to enable the tracing detail needed in the guestbook frontend application:

```
spring.cloud.gcp.trace.enabled=true
spring.sleuth.sampler.probability=1.0
spring.sleuth.scheduled.enabled=false
```

spring.cloud.gcp.trace.enabled=true
spring.sleuth.sampler.probability=1.0
spring.sleuth.scheduled.enabled=false

The Demo Java Microservices Application

❑ Working with Cloud Trace

Task 5. Set up a service account

In this task, you create a service account with permissions to propagate trace data to Cloud Trace.

1. Click on the `Open Terminal` icon and then in Cloud Shell enter the following commands to create a service account specific to the guestbook application:

```
gcloud iam service-accounts create guestbook
```

2. Add the Editor role for your project to this service account:

```
export PROJECT_ID=$(gcloud config list --format
'value(core.project)')
gcloud projects add-iam-policy-binding ${PROJECT_ID} \
  --member
serviceAccount:guestbook@${PROJECT_ID}.iam.gserviceaccou
\
  --role roles/editor
```

3. Generate the JSON key file to be used by the application to identify itself using the service account:

```
gcloud iam service-accounts keys create \
    ~/service-account.json \
    --iam-account
guestbook@${PROJECT_ID}.iam.gserviceaccount.com
```

This command creates service account credentials that are stored in the `$HOME/service-account.json` file.

```
gcloud iam service-accounts create guestbook
export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
gcloud projects add-iam-policy-binding ${PROJECT_ID} \
  --member serviceAccount:guestbook@${PROJECT_ID}.iam.gserviceaccount.com \
  --role roles/editor
gcloud iam service-accounts keys create \
    ~/service-account.json \
    --iam-account guestbook@${PROJECT_ID}.iam.gserviceaccount.com
```

The Demo Java Microservices Application

❑ Working with Cloud Trace

Task 6. Run the application locally with your service account

In this task you start both the frontend application and backend service application using the additional `spring.cloud.gcp.credentials.location` property to specify the location of the service account credential that you created.

1. Change to the `guestbook-service` directory:

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

4. Change to the `guestbook-frontend` directory:

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

5. Start the guestbook frontend application:

```
./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud \
  -
Dspring.cloud.gcp.credentials.location=file:///$HOME/ser
account.json"
```

6. Post a message using the Cloud Shell web preview of the application to generate trace data.

2. Start the guestbook backend service application:

```
./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud \
  -
Dspring.cloud.gcp.credentials.location=file:///$HOME/ser
account.json"
```

This takes a minute or two to complete and you should wait until you see that the GuestbookApplication is running:

```
Started GuestbookApplication in 20.399 seconds (JVM
running...)
```

3. Open a new Cloud Shell tab by clicking the Add icon to the right of the title of the current Cloud Shell tab.

```
cd ~/guestbook-service

cd ~/guestbook-frontend

./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-
Dspring.profiles.active=cloud \
  -
Dspring.cloud.gcp.credentials.location=file:///$HOME/
service-account.json"
```

The Demo Java Microservices Application

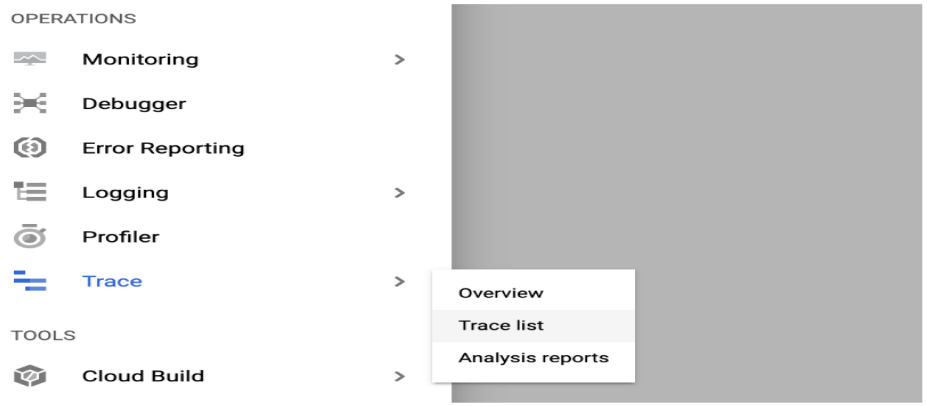
❑ Working with Cloud Trace

Task 7. Examine the trace

In this task you examine the trace. The trace has been generated and propagated by Spring Cloud Sleuth. After a few seconds it'll be propagated to Cloud Trace.

1. Open the Google Cloud console browser tab.

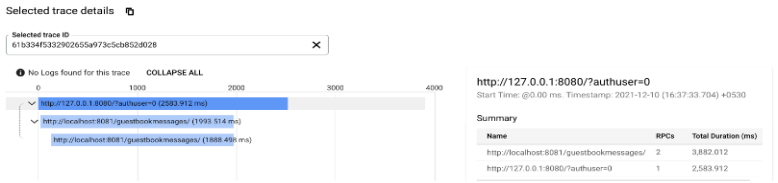
2. In the Navigation Menu navigate to **Trace > Trace List** in the **Operations** section.



3. At the top of the window, set the time range to 1 hour.

4. Turn **Auto reload** on. New trace data may take up to 30 seconds to appear.

5. Click the blue dot to view trace detail.



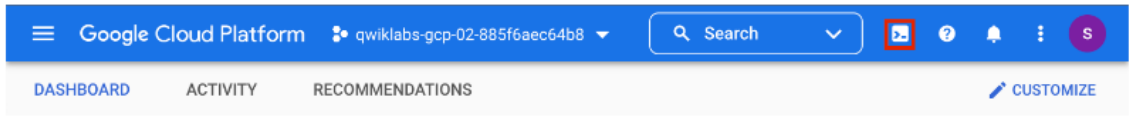
The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Pub/Sub is a fully managed, real-time messaging service that enables you to send and receive messages between independent applications.

For this part, we will enhance our application to implement a message handling service with Pub/Sub so that it can publish a message to a topic that can then be subscribed and processed by other services.

1. In Cloud console, on the top right toolbar, click the Open Cloud Shell button.



2. Click **Continue**.

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

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.

You can switch between Cloud Shell and the code editor by using `Open Editor` and `Open Terminal` icon as required.

6. Make the Maven wrapper scripts executable:

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

Now you're ready to go!

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)')
```

4. Verify that the demo application files were created:

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

5. Copy the application folders to Cloud Shell:

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

export PROJECT_ID=$(gcloud config list --format 'value(core.project)')

gsutil ls gs://$PROJECT_ID

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

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

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 2. Enable Pub/Sub API

1. In Cloud Shell, enable the Pub/Sub API:

```
gcloud services enable pubsub.googleapis.com
```

gcloud services enable pubsub.googleapis.com

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 3. Create a Pub/Sub topic

In this task, you create a Pub/Sub topic to which you will send a message:

1. Use `gcloud` to create a Pub/Sub topic:

```
gcloud pubsub topics create messages
```

gcloud pubsub topics create messages

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 4. Add Spring Cloud GCP Pub/Sub starter

In this task, you update the guestbook frontend application's `pom.xml` file to include the Spring Cloud GCP starter for Pub/Sub in the dependency section.

1. Open the Cloud Shell code editor.

2. In the code editor, open `~/guestbook-frontend/pom.xml`.

3. Insert the following new dependency at the end of the `<dependencies>` section, just before the closing `</dependencies>` tag:

```
        <dependency>

<groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-
pubsub</artifactId>
        </dependency>
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-gcp-starter-pubsub</artifactId>
</dependency>
```

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 5. Publish a message

In this task, you use the `PubSubTemplate` bean in Spring Cloud GCP to publish a message to Pub/Sub. This bean is automatically configured and made available by the starter. You add `PubSubTemplate` to `FrontendController`.

1. Open `guestbook-frontend/src/main/java/com/example/frontend/FrontendControlle` in the Cloud Shell code editor.

2. Add the following statement immediately after the existing `import` directives:

```
import org.springframework.cloud.gcp.pubsub.core.*;
```

3. Insert the following statement between the lines `private GuestbookMessagesClient client;` and `@Value("${greeting:Hello}")`:

```
@Autowired
private PubSubTemplate pubSubTemplate;
```

4. Add the following statement inside the if statement to process messages that aren't null or empty, just below the comment `// Post the message to the backend service`:

```
pubSubTemplate.publish("messages", name + ": " + message);
```

```
import org.springframework.cloud.gcp.pubsub.core.*;

@Autowired
  private PubSubTemplate pubSubTemplate;

pubSubTemplate.publish("messages", name + ": " + message);
```

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 6. Test the application in the Cloud Shell

In this task, you run the application in the Cloud Shell to test the new Pub/Sub message handling code.

1. In Cloud Shell, change to the `guestbook-service` directory:

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

2. Run the backend service application:

```
./mvnw spring-boot:run -Dspring-boot.run.jvmArguments="-Dspring.profiles.active=cloud"
```

The backend service application launches on port 8081.This takes a minute or two to complete and you should wait until you see that the GuestbookApplication is running.

3. Open a new Cloud Shell session tab to run the frontend application by clicking the plus (+) icon to the right of the title tab for the initial Cloud Shell session.

4. Change to the `guestbook-frontend` directory:

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

5. Start the frontend application with the `cloud` profile:

```
./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

6. Open the Cloud Shell web preview and post a message.

The frontend application tries to publish a message to the Pub/Sub topic. You will check if this was successful in the next task.

./mvnw spring-boot:run -Dspring-boot.run.jvmArguments="-Dspring.profiles.active=cloud“

./mvnw spring-boot:run -Dspring.profiles.active=cloud

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 7. Create a subscription

Before subscribing to a topic, you must create a subscription. Pub/Sub supports pull subscription and push subscription. With a pull subscription, the client can pull messages from the topic. With a push subscription, Pub/Sub can publish messages to a target webhook endpoint.

A topic can have multiple subscriptions. A subscription can have many subscribers. If you want to distribute different messages to different subscribers, then each subscriber needs to subscribe to its own subscription. If you want to publish the same messages to all the subscribers, then all the subscribers must subscribe to the same subscription.

Pub/Sub messages are delivered "at least once." Thus, you must deal with idempotence and you must deduplicate messages if you cannot process the same message more than once.

In this task, you create a Pub/Sub subscription and then test it by pulling messages from the subscription before and after using the frontend application to post a message.

1. Open a new Cloud Shell tab.

2. Create a Pub/Sub subscription:

```
gcloud pubsub subscriptions create messages-subscription-1 \
  --topic=messages
```

3. Pull messages from the subscription:

```
gcloud pubsub subscriptions pull messages-subscription-1
```

The `pull messages` command should report 0 items.

The message you posted earlier does not appear, because the message was published before the subscription was created.

4. Return to the frontend application, post another message, and then pull the message again:

```
gcloud pubsub subscriptions pull messages-subscription-1
```

The message appears. The message remains in the subscription until it is acknowledged.

5. Pull the message again and remove it from the subscription by using the auto-acknowledgement switch at the command line:

```
gcloud pubsub subscriptions pull messages-subscription-1 --auto-ack
```

gcloud pubsub subscriptions create messages-subscription-1 \
  --topic=messages

gcloud pubsub subscriptions pull messages-subscription-1 --auto-ack

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

Task 8. Process messages in subscriptions

In this task, you use the Spring `PubSubTemplate` to listen to subscriptions.

1. In Cloud Shell, generate a new project from Spring Initializr.

```
cd ~
curl https://start.spring.io/starter.tgz \
  -d dependencies=web,cloud-gcp-pubsub \
  -d bootVersion=2.4.6.RELEASE \
  -d baseDir=message-processor | tar -xzvf -
```

This command generates a new Spring Boot project with the Pub/Sub starter preconfigured. The command also automatically downloads and unpacks the project into the `message-processor` directory structure.

2. Open `~/message-processor/pom.xml` to verify that the starter dependencies were automatically added.

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.google.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-
pubsub</artifactId>
        </dependency>
    </dependencies>
```

3. To write the code to listen for new messages delivered to the topic, open `~/message-processor/src/main/java/com/example/demo/DemoApplication.java` in the Cloud Shell code editor.

4. Add the following `import` directives below the existing `import` directives:

```
import org.springframework.context.annotation.Bean;
import org.springframework.boot.ApplicationRunner;
import com.google.cloud.spring.pubsub.core.*;
```

The Demo Java Microservices Application

❑ Messaging with Pub/Sub

5. Add the following code block to the class definition for `DemoApplication`, just above the existing definition for the main method:

```
    @Bean
    public ApplicationRunner cli(PubSubTemplate
pubSubTemplate) {
        return (args) -> {
            pubSubTemplate.subscribe("messages-
subscription-1",
                (msg) -> {

System.out.println(msg.getPubsubMessage()
                        .getData().toStringUtf8());
                    msg.ack();
                });
        };
    }
```

We added the Web starter simply because it's much easier to put Spring Boot application into daemon mode, so that it doesn't exit immediately. There are other ways to create a Daemon, e.g., using a CountDownLatch, or create a new Thread and set the daemon property to true. But since we are using the Web starter, make sure that the server port is running on a different port to avoid port conflicts.

6. Add this line to change the port on `message-processor/src/main/resources/application.properties`:

```
server.port=${PORT:9090}
```

7. Return to the Cloud Shell tab for the message processor to listen to the topic.

```
cd ~/message-processor
./mvnw -q spring-boot:run
```

8. Open the browser with the frontend application, and post a few messages.

9. Verify that the Pub/Sub messages are received in the message processor.

The new messages should be displayed in the Cloud Shell tab where the message processor is running, as in the following example:

```
... [main] com.example.demo.DemoApplication         : Started
DemoApplication...
Ray: Hey
Ray: Hello!
```

Conclusion

We have learned how to create Spring Boot microservice on Google Cloud Platform(GCP) as well as how to apply scalability to our microservices using Spring boot, Spring Cloud and GCP. The links of our resources and GitHub repository are attached below

https://www.coursera.org/learn/google-cloud-java-spring/home/welcome

https://github.com/Dilane-Kamga/Scalable-Microservice-Spring-boot-and-Spring-cloud.git

MERCI

Pour votre attention