

Google Cloud Native with Spring Boot

Self-link: bit.ly/spring-gcp-lab

Slides: [PDF](#)

Author: Ray Tsang [@saturnism](#)

Overview

Duration: 5:00

In this lab, you'll quickly create several microservices with Spring Boot and adopt some of the best practices around tracing, configuration management, and integration with other parts of the system with integration patterns. To accomplish this, we'll use Spring Cloud Sleuth, Zipkin, Config Server, and Spring Integration to build a messaging pipeline.

This is great when running applications on-premises. But what do you do when you move to the cloud? Cloud Native is not just about migrating from bare-metal workload to VM-based workload! Cloud Native applications can adapt to the new environments that are fully managed and require less to no manual operation.

From the lab, you'll learn how to replace the external dependencies that you would need to maintain and operate yourself with fully managed services on Google Cloud Platform. Through the use of the new Spring Cloud GCP starters, we can quickly replace RDBMS like MySQL with CloudSQL, messaging like RabbitMQ with Pub/Sub, distributed trace stores like Zipkin with Stackdriver Trace, and centralized Config Server with Runtime Config.

However, the database is usually the constraining performance bottleneck. We'll also go over the different database options on Google Cloud Platform to help you eliminate these bottlenecks.

What is your experience level with Google Cloud Platform?

- I have just heard of Google Cloud Platform
- I have played around with Google Cloud Platform
- I use Google Cloud Platform in production

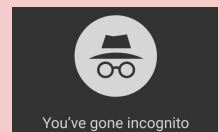
Lab 0 - Initial setup

Duration: 20:00

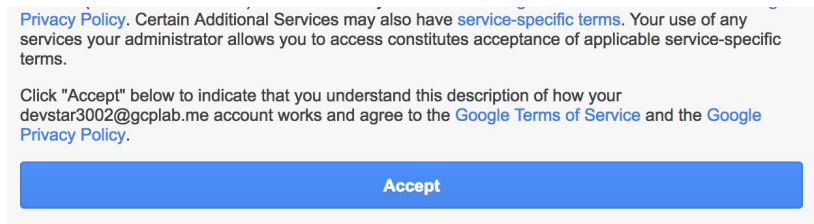
Task 1 - Logging In

1. The instructor will provide you with a temporary username / password to sign in to the Google Cloud Platform Console.

Important: To avoid conflicts with your personal account, please open a new incognito window for the rest of this lab.



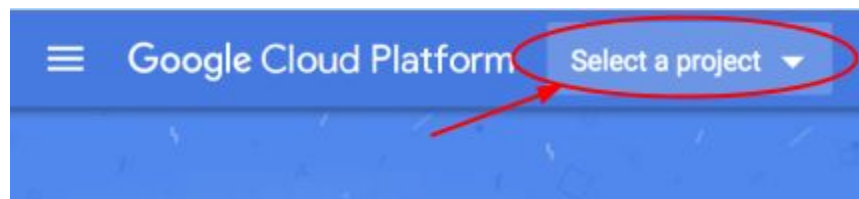
2. Sign in to the Google Cloud Platform Console: <https://console.cloud.google.com/> with the provided credentials. In **Welcome to your new account** dialog, click **Accept**.



3. If you see a top bar with **Sign Up for Free Trial** - DO NOT SIGN UP FOR THE FREE TRIAL. Click **Dismiss** since you'll be using a pre-provisioned lab account. If you are doing this on your own account, then you may want the free trial.



4. Click **Select a project**.



5. In the **All** tab, click on your project:



You should see the Project Dashboard:

DASHBOARD
ACTIVITY
CUSTOMIZE

Project info

Project name
Spring IO Barcelona 2401

Project ID
springio18-bcn-2401

Project number
1056330239641

→ Go to project settings

Resources

This project has no resources

Trace

No trace data from the past 7 days

API APIs

Requests (requests/sec)

→ Go to APIs overview

Google Cloud Platform status

All services normal

→ Go to Cloud status dashboard

Billing

Estimated charges USD \$0.00
For the billing period May 1 – 23, 2018

→ View detailed charges

Error Reporting

No sign of any errors. Have you set up Error Reporting?

→ Learn how to set up Error Reporting

Task 2 - Cloud Shell

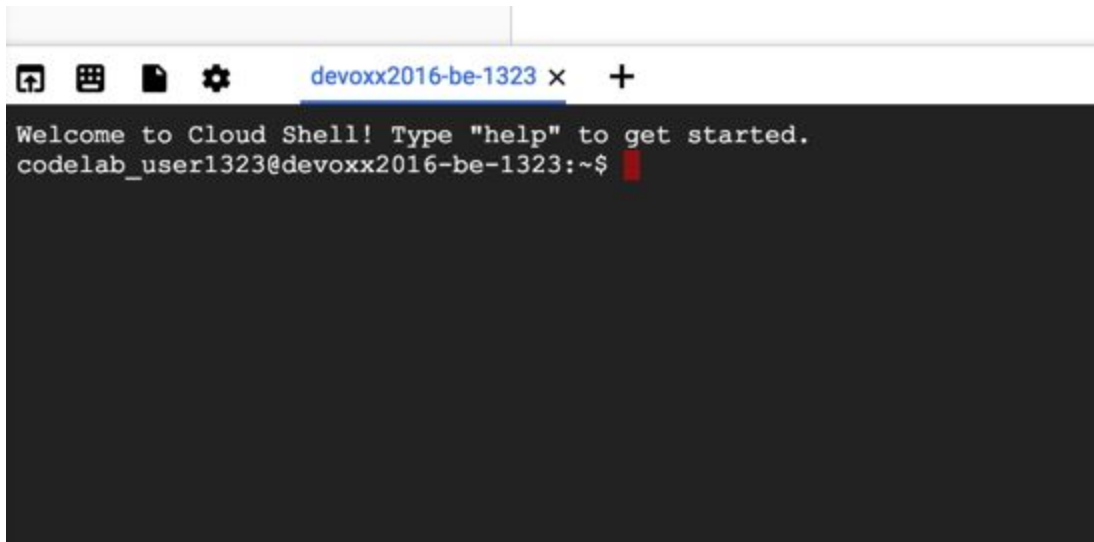
You will do most of the work from the [Google Cloud Shell](#), a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (`docker`, `gcloud`, `kubectl` and others) and offers a persistent 5GB home directory. Open the Google Cloud Shell by clicking on the icon on the top right of the screen:



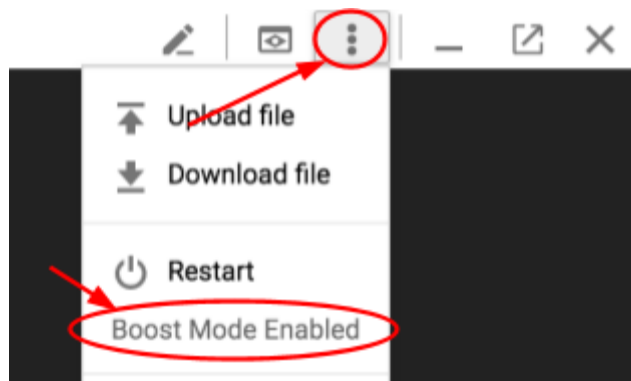
1. When prompted, click **Start Cloud Shell**:



You should see the shell prompt at the bottom of the window:



2. Check to see if Cloud Shell's Boost mode is Enabled.



3. If not, then enable **Boost Mode** for Cloud Shell.

Note: When you run `gcloud` on your own machine, the config settings will be persisted across sessions. But in Cloud Shell, you will need to set this for every new session / reconnection.

Task 3 - Solutions are on GitHub

If you fall behind, the solutions for each of the sections are on GitHub:

<https://github.com/saturnism/spring-cloud-gcp-guestbook>

Lab 1 - Bootstrap the Frontend and Backend

Duration: 20:00

You should perform all of the lab instructions directly in Cloud Shell.

Task 1 - Bootstrap Backend

1. To save time, you can simply copy from solution's `1-bootstrap` directory.

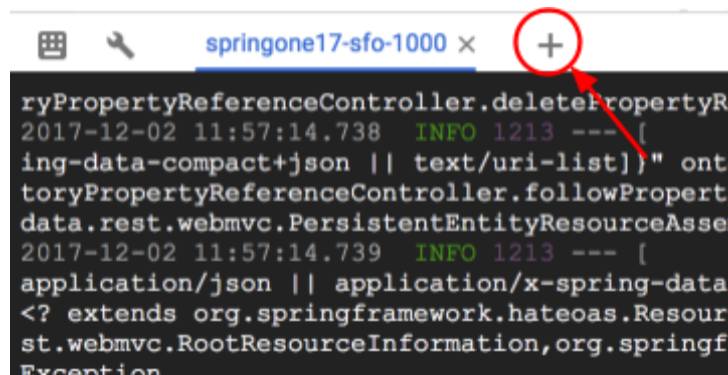
```
$ cd ~/
$ git clone https://github.com/saturnism/spring-cloud-gcp-guestbook.git
$ cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-service \
  ~/guestbook-service
```

Task 2 - Run Backend Locally

1. Test it out.

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

2. Open a new Cloud Shell session tab.



3. While the service is still running, open a new tab, and test out the service.

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

4. You can post a new message.

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

5. You can also list all the messages.

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

Task 3 - Bootstrap Frontend

1. To save time, you can simply copy from the solution's 1-bootstrap directory, and skip to Run it locally section.

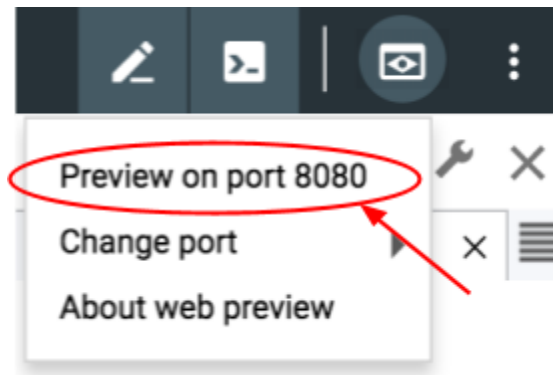
```
$ cp -a ~/spring-cloud-gcp-guestbook/1-bootstrap/guestbook-frontend \
~/guestbook-frontend
```

Taks 4 - Run Frontend Locally

1. Test it out.

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

2. This should launch the frontend application on port 8080. Use Cloud Shell's web preview for port 8080.



This will open a new browser tab.

Guestbook

Your name:

Message:

3. Try to post the name and the message. Once done, you should see the messages listed below.

Guestbook

Your name:

Message:

Post

Ray Hello

Ray Hi

Not bad! That's a quick way to put together a simple application composed of a microservices backend, and a frontend consuming it. Next you'll see how you can productionalize it.

4. In a new shell tab, you can list all the messages you added via the backend API.

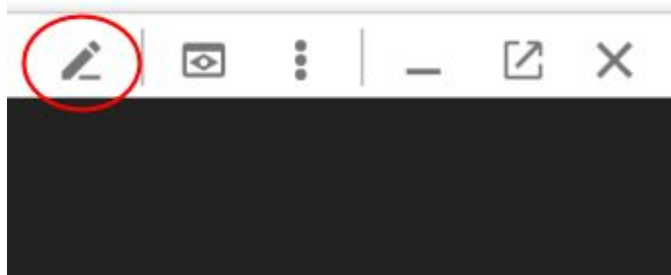
```
$ curl -s http://localhost:8081/guestbookMessages
```

5. Or, use `jq` to parse the JSON text and print out only the messages.

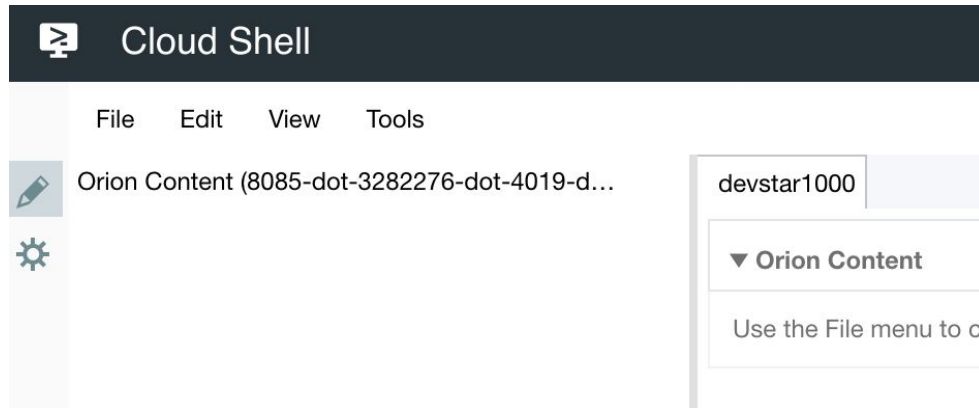
```
$ curl -s http://localhost:8081/guestbookMessages \
| jq -r '._embedded.guestbookMessages[] | {name: .name, message: .message}'
```

Task 5 - Using the Cloud Code Editor (Beta)

1. If you are not familiar with a text-based editor, like `vi`, `nano`, or `emacs`, you can use the web-based code editor. Click on the Code Editor icon on the right hand side of the Cloud Shell.



This will launch a new browser tab with the Eclipse Orion editor.



Note: To avoid extra setup, we'll mostly use this editor for the code lab, and all the commands will be entered within Cloud Shell.

Lab 2 - Cloud SQL

Duration: 30:00

Rather than maintaining your own MySQL instance, in the cloud, you should use managed services as much as possible to reduce operation overhead and increase reliability. Google Cloud Platform has a managed MySQL and PostgreSQL service called CloudSQL.

Task 1 - Check if Cloud SQL has been pre-provisioned

Some versions of this lab may have pre-provisioned a Cloud SQL instance. You can verify by using the `gcloud` or navigating to the SQL section of the Cloud Console. If you do see a pre-provisioned database instance skip ahead to Task 4, otherwise proceed to Task 2.

1. Enable the Cloud SQL Admin API.

```
$ gcloud services enable sqladmin.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the
Operation details:
  gcloud services operations describe operations/...
```

2. If you see a `guestbook` Cloud SQL instance pre-provisioned, then skip to [Task 3](#).

For example:

```
$ gcloud sql instances list
```

NAME	DATABASE_VERSION	LOCATION	TIER	ADDRESS	STATUS
guestbook	MYSQL_5_7	us-central1-b	db-n1-standard-1	123.45.67.8	RUNNABLE

3. If you don't see a `guestbook` Cloud SQL instance pre-provisioned, then proceed to [Task 2](#).

For example:

```
$ gcloud sql instances list
Listed 0 items.
```

Task 2 - Create a new Cloud SQL Instance if not pre-provisioned

Note: Please skip this step if Task 1 or the instructor instructed you to skip.

1. Provision a new CloudSQL instance (this will take some time).

```
$ gcloud sql instances create guestbook --region=us-central1
Creating Cloud SQL instance...done.
Created [...].
```

NAME	DATABASE_VERSION	REGION	TIER	ADDRESS	STATUS
guestbook	MYSQL_5_6	us-central1	db-n1-standard-1	92.3.4.5	RUNNABLE

Task 3 - Create a Database in the Cloud SQL Instance

1. Create a new `messages` database within the MySQL instance.

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

Task 4 - Connect to CloudSQL and create the schema

CloudSQL, by default, is not accessible via any public IP addresses. There are several different ways to connect to it:

- Using a local CloudSQL proxy
- Use `gcloud` to connect via a CLI client
- From Java application, using the MySQL JDBC driver with an SSL Socket Factory for secured connection

1. Use `gcloud` CLI to connect to the database. This will temporarily whitelist the IP address for connection. The `root` password is empty by default.

```
$ gcloud sql connect guestbook
Whitelisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password: [PRESS ENTER]
...
```

Note: For security reasons, Cloud SQL public IP is not accessible by anyone unless explicitly whitelisted. The command line can automatically & temporarily whitelist your incoming connection. It takes a minute to complete whitelisting and connect.

2. List the databases.

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| messages |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.04 sec)
```

3. Switch to the `messages` database and create the table.

```
mysql> use messages;
Database changed
mysql> 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)
);
mysql> exit
```

Task 5 - Add Spring Cloud GCP CloudSQL Starter

From a Java application, you can consume the CloudSQL instance using the JDBC driver. However, the JDBC driver configuration can be a little bit more complicated than most due to additional security that Google Cloud Platform put in place.

To make this configuration we created a Spring Cloud GCP project that can easily auto-configure your Spring Boot applications to consume our services, including CloudSQL.

1. Update the Guestbook Service's `pom.xml` to import the Spring Cloud GCP BOM and also the Spring Cloud GCP Cloud SQL Starter. This involves adding the milestone repository to use our latest Release Candidate.

guestbook-service/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
```

```

        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-gcp-dependencies</artifactId>
        <version>1.1.0.M1</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencies>
</dependencyManagement>

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

...

<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/libs-milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
</project>

```

Task 6 - Disable CloudSQL in the default profile

1. For local testing, you can continue to use a local database or an embedded database. For example, this lab uses embedded HSQL database for local runs. You can disable CloudSQL starter in the default profile.

Update the existing `application.properties`.

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

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

Task 7 - Configure a Cloud Profile

When deploying this into the cloud, you'll want to use the production managed CloudSQL instance. To do this, we'll use Spring's configuration profile and create a new `cloud` profile.

1. Find the Instance Connection name using the command line.

```
$ gcloud sql instances describe guestbook --format='value(connectionName)'  
your_project:us-central1:guestbook ← whole string is instance connection name
```

2. Create a new `application-cloud.properties`.

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

Task 8 - Configuring Connection Pool

You can configure the JDBC connection pool just like you do with any other Spring Boot applications using the `spring.datasource.*` configuration properties.

1. Reduce the connection pool size.

`guestbook-service/src/main/resources/application-cloud.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  
  
spring.datasource.hikari.maximum-pool-size=5
```

Task 9 - Run it locally

1. Make sure you are in `guestbook-service` directory, or kill the existing `guestbook-service` application and do this in that Cloud Shell tab.

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

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

```
$ ./mvnw -q test
```

3. Stop the Guestbook Service that's already running, and restart it with the `cloud` profile.

```
$ ./mvnw spring-boot:run -Dserver.port=8081 -Dspring.profiles.active=cloud
```

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

```
... First Cloud SQL connection, generating RSA key pair.
... Obtaining ephemeral certificate for Cloud SQL instance
[springone17-sfo-1000:us-ce
... Connecting to Cloud SQL instance [...:us-central1:guestbook] on ...
... Connecting to Cloud SQL instance [...:us-central1:guestbook] on ...
... Connecting to Cloud SQL instance [...:us-central1:guestbook] on ...
...
```

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. You can also use CloudSQL client to validate.

```
$ gcloud sql connect guestbook
Whitelisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password: [PRESS ENTER]
...
mysql> use messages
mysql> select * from guestbook_message;
+----+-----+-----+-----+
| id | name | message          | image_uri |
+----+-----+-----+-----+
|  1 | Ray  | Hello CloudSQL  | NULL      |
+----+-----+-----+-----+
1 row in set (0.04 sec)
mysql> exit;
```

Lab 3 - Runtime Configuration

Duration: 30:00

There are a number of ways to configure your microservices application. Typically you end up with additional Configuration Server that you'll have to make highly available and manage yourself. Google Cloud Platform has a Runtime Configuration service, where you can store arbitrary configuration key/value pairs.

Spring Cloud GCP has a configuration starter that interoperates well with Spring Cloud Config facility, so that you can configure your existing application easily without a manually managed config server.

The greeting message produced by the frontend is configurable via `greeting` property. We can externalize this into Google Cloud Platform runtime configuration.

In the code:

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java, we are reading `${greeting}` variable, be defaulted to "Hello". We can use Runtime Configuration to override this value.

Task 1 - Add Spring Cloud GCP Config Starter

1. Update the Frontend's `pom.xml` to import the Spring Cloud GCP BOM. This involves adding the milestone repository (since the version we use is in this Maven repository). Then include the starter in dependency.

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
    ...

    <dependencies>
        ...
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-config</artifactId>
        </dependency>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>20.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            ...
            <!-- add after existing dependency management -->
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-gcp-dependencies</artifactId>
                <version>1.1.0.M1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    ...
    <repositories>
        <repository>
            <id>spring-milestones</id>
```

```
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
</project>
```

Task 2 - Disable Runtime Config in the default profile

For local testing, you might not want to connect to a remote Runtime Config server and choose to use local configurations.

1. Create a new `bootstrap.properties` that disables the Spring Cloud GCP starter.

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

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

Task 3 - Configure a Cloud Profile

When deploying this into the cloud, you'll want to use the production configurations. To do this, we'll use Spring's configuration profile and create a new `cloud` profile.

1. Create a new `bootstrap-cloud.properties` with the Spring Cloud Config bootstrapping configuration.

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

```
spring.cloud.gcp.config.enabled=true
spring.cloud.gcp.config.name=frontend
spring.cloud.gcp.config.profile=cloud
```

2. Allow access to call the Spring Actuator endpoint so you can dynamically refresh the configuration.

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

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

Note: The combination of

`${spring.cloud.gcp.config.name}_${spring.cloud.gcp.config.profile}` forms `frontend_cloud`, which is the name of the Runtime Configuration we will create in the next step.

Task 4 - Add Refresh Scope to Frontend Controller

1. Add `@RefreshScope` annotation to the `FrontendController`.

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java

```
package com.example.frontend;

...
import org.springframework.cloud.context.config.annotation.RefreshScope;

@RefreshScope
@Controller
@SessionAttribute("name")
public class FrontendController {
    ...
}
```

Task 5 - Create a new Runtime Configuration

1. Enable the Runtime Configuration API.

```
$ gcloud services enable runtimeconfig.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the
Operation details:
  gcloud services operations describe operations/...
```

2. Create a new Runtime Configuration for the Frontend's `cloud` profile.

```
$ gcloud beta runtime-config configs create frontend_cloud
Created [https://runtimeconfig.googleapis.com/v1beta1/projects/...].
```

3. Set the configuration value.

```
$ gcloud beta runtime-config configs variables set greeting \
  "Hi from Runtime Config" \
  --config-name frontend_cloud
```

4. To see all the variables in the runtime configuration.

```
$ gcloud beta runtime-config configs variables list --config-name=frontend_cloud
```

5. To see the value of a specific variable.


```
$ gcloud beta runtime-config configs variables \
  get-value greeting --config-name=frontend_cloud
```

Task 6 - Run it locally

1. Make sure you are in `guestbook-frontend` directory (or, switch back to the `guestbook-frontend` shell tab, and stop the process, and restart the instance).

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

2. To test this all works, first, run a test with the default profile.

```
$ ./mvnw -q test
```

3. Stop the already running Guestbook Frontend and restart it with the `cloud` profile.

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

4. Use Cloud Shell preview to browse to the frontend.

5. Enter **name** and **message**, and click **Post** to trigger the Hello response.

6. Validate that the returned greeting message is now coming from Cloud Runtime Config.

Guestbook

Your name:

Message:

Post

Hi from Runtime Config! Ray

Ray

Hello CloudSQL

Ray

Hello

Note: If you use Spring Cloud Actuator, then you can actually refresh and reload configuration on the fly. See [sample code](#) for how that works.

Task 7 - Update Configuration and Refresh

1. Update the `greeting` configuration value.

```
$ gcloud beta runtime-config configs variables set greeting \  
  "Hi from Updated Config" \  
  --config-name frontend_cloud
```

2. In a new terminal session, use Spring Actuator to reload the configuration from the server.

```
$ curl -XPOST http://localhost:8080/actuator/refresh \  
["greeting"]
```

The configuration should be reloaded!

3. Post another message to the Guestbook application through the frontend, you should see the updated greeting response.

Guestbook

Your name:

Message:

Hi from Updated Config Ray

4. You can use Spring Actuator to see the current configuration values, and [other information too](#).

```
$ curl http://localhost:8080/actuator/configprops | jq
```

The configuration should be reloaded!

Lab 4 - Stackdriver Trace

Duration: 30:00

In a microservices architecture, you need distributed tracing to get better observability for complicated service calls. E.g., when service A calls B that calls C, which service is having issue? In Spring Cloud, you can add

distributed tracing easily using Spring Cloud Sleuth. This typically requires you to run and operate your own Zipkin backend.

Spring Cloud GCP provides a starter that can interoperate with Spring Cloud Sleuth, but forwards the trace request to Stackdriver Trace instead!

Task 1 - Enable Stackdriver Trace API

1. In a new shell tab, enable Stackdriver Trace API first In order to use Stackdriver Trace to store your trace data.

```
$ gcloud services enable cloudtrace.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the Operation
details:
  gcloud services operations describe operations/...
```

Task 2 - Add Spring Cloud GCP Trace Starter

1. To turn this on is easy - simply add the starter into your `pom.xml`.

guestbook-service/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-starter-trace</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-starter-trace</artifactId>
    </dependency>
  </dependencies>
  ...
```

```
</project>
```

Task 3 - Configure Applications

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

1. For testing purposes, we'll disable trace in the local profile.

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

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

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

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

2. For the cloud profile, we'll enable trace sampling for 100% of the requests.

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

```
...  
spring.cloud.gcp.trace.enabled=true  
spring.sleuth.sampler.probability=1  
spring.sleuth.web.skipPattern=(^cleanup.*|.+favicon.*)
```

3. Create a new Cloud configuration for Guestbook Frontend:

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

```
spring.cloud.gcp.trace.enabled=true  
spring.sleuth.sampler.probability=1  
spring.sleuth.web.skipPattern=(^cleanup.*|.+favicon.*)
```

Task 4 - Setup a Service Account

For this lab, you'll need to use a service account with the proper permissions to propagate Trace data to Stackdriver Trace.

1. Create a service account specific to the Guestbook application.

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')  
$ gcloud iam service-accounts create guestbook
```

2. Add the Project Editor role to this service account.

```
$ gcloud projects add-iam-policy-binding ${PROJECT_ID} \
  --member serviceAccount:guestbook@${PROJECT_ID}.iam.gserviceaccount.com \
  --role roles/editor
```

Warning: This creates a service account with the Project Editor role. In your production environment, you should only assign roles and permissions that the application actually needs.

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
```

Note: This should create a service account credentials file and stored in the `~/service-account.json` file. Treat this file as your own username/password. Do not share this in the public!

Task 5 - Run it locally with service account

To test, restart both applications, but with the additional `spring.cloud.gcp.credentials.location` property to specify the location of the service account credential you created.

1. Stop the existing Guestbook Service and restart it.

```
$ cd ~/guestbook-service
$ ./mvnw spring-boot:run -Dserver.port=8081 -Dspring.profiles.active=cloud \
  -Dspring.cloud.gcp.credentials.location=file:///~/service-account.json
```

2. Stop the existing Guestbook Frontend and restart it.

```
$ cd ~/guestbook-frontend
$ ./mvnw spring-boot:run -Dspring.profiles.active=cloud \
  -Dspring.cloud.gcp.credentials.location=file:///~/service-account.json
```

3. Make a request from Cloud Shell web preview.

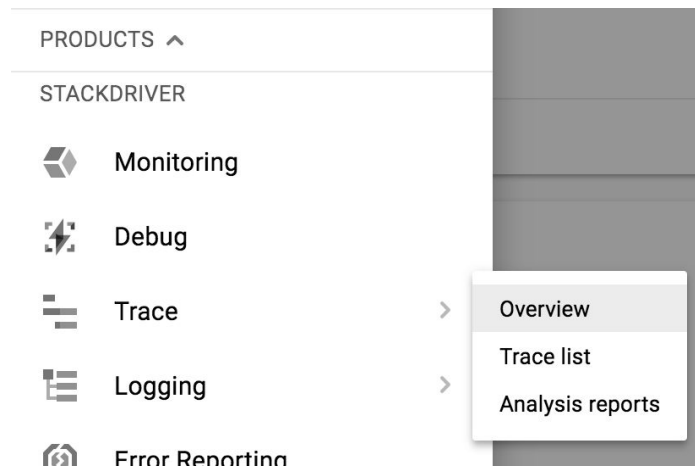
4. Or, make a couple of requests from Cloud Shell.

```
$ curl http://localhost:8080
```

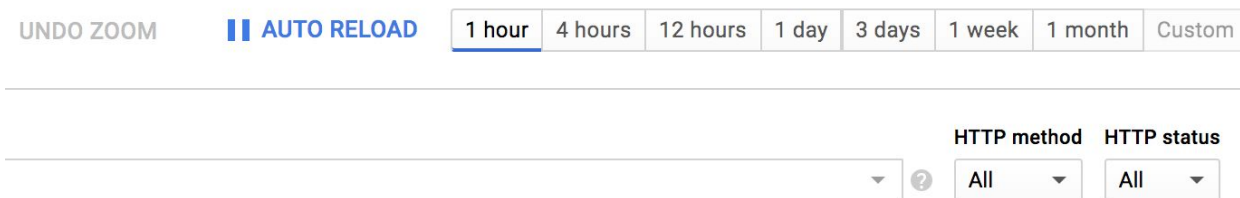
Task 6 - Examine the traces

The trace would've been generated and propagated by Spring Cloud Sleuth, and in a few seconds or so, it'll be propagated to Stackdriver Trace.

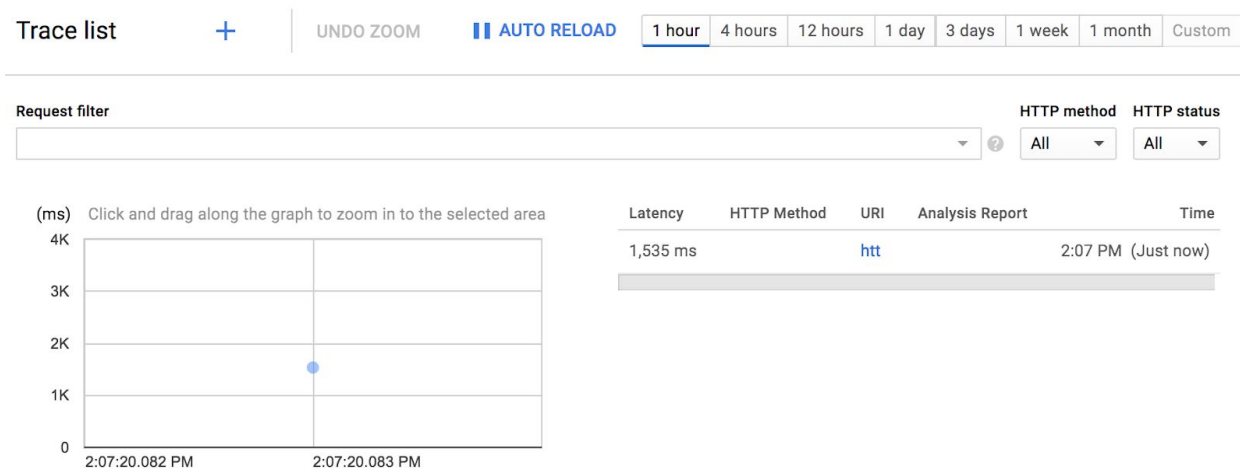
1. In a new browser tab, from the Google Cloud Platform console, navigate to **Stackdriver** → **Trace** → **Trace List**. In here, you should see the traces for the requests you've made.



2. On the top, narrow down the time range to 1 hour. By default, **Auto Reload** is on. So as trace data arrive, it should show up in the console!

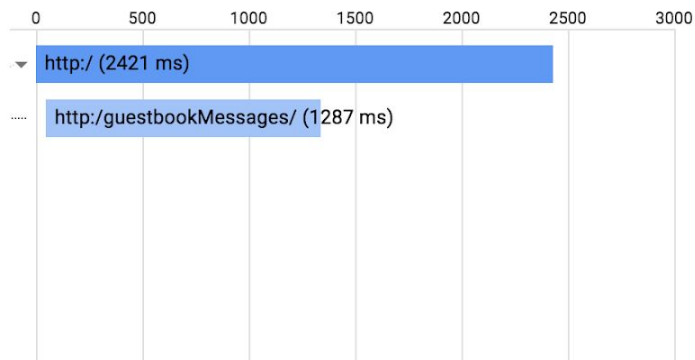


The trace data should show up in ~30 seconds or so.



3. Click the **blue** dot to see trace detail.

Timeline



@0 ms

`http:/`

Summary

Name	RPCs	Total Duration (ms)
<code>http:/</code>	1	2421
<code>http:/guestbookMessages/</code>	1	1287

Lab 5a - Messaging with Pub/Sub

Duration: 1:00:00

Let's enhance our application so that it can publish a message to a topic that can then be subscribed and processed by other services.

Task 1 - Enable Pub/Sub API

1. Enable the Pub/Sub API.

```
$ gcloud services enable pubsub.googleapis.com
Waiting for async operation operations/... to complete...
Operation finished successfully. The following command can describe the
Operation details:
  gcloud services operations describe operations/...
```

Task 2 - Create a new Topic

1. Create a new topic to send the message to.

```
$ gcloud pubsub topics create messages
```

Task 3 - Add Spring Cloud GCP Pub/Sub Starter

1. In the Guestbook Frontend, add the Pub/Sub starters.

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  ...
  <dependencies>
```

```

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

```

Task 4 - Publish Message

The simplest way to publish a message to Pub/Sub using Spring Cloud GCP is to use the `PubSubTemplate` bean. This bean is automatically configured and made available by the starter.

1. Add `PubSubTemplate` to `FrontendController` and use it to publish a message.

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java

```

package com.example.frontend;

...

// Add imports
import org.springframework.cloud.gcp.pubsub.core.*;

@Controller
@SessionAttributes("name")
public class FrontendController {
    @Autowired
    private GuestbookMessagesClient client;

    @Autowired
    private PubSubTemplate pubSubTemplate;

    @Value("${greeting:Hello}")
    private String greeting;

    @GetMapping("/")
    public String index(Model model) {
        ...
    }

    @PostMapping("/post")
    public String post(@RequestParam String name, @RequestParam String
message, Model model) {
        model.addAttribute("name", name);
        if (message != null && !message.trim().isEmpty()) {
            // Post the message to the backend service
            ...
        }
    }
}

```



```
        pubSubTemplate.publish("messages", name + ": " + message);
    }
    return "redirect:/";
}
}
```

Task 5 - Run it locally

1. Restart the Guestbook Frontend.

```
$ cd ~/guestbook-frontend
$ ./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

2. Open Cloud Shell web preview and post a message. This will then publish a message to the Pub/Sub topic.

Task 6 - Create a Subscription

To subscribe to a topic, you need to create a subscription first. Pub/Sub supports pull subscription and push subscription. With a pull subscription, client can actively pull messages from the topic. With a push subscription, Pub/Sub can actively 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 around to different subscribers, then all the subscribers must be subscribing to the same subscription. If you want to publish the same messages to all the subscribers, then each subscriber needs to subscribe to its own subscription.

Pub/Sub delivery is at-least once. Hence, you must deal with idempotency and/or de-duplicate messages if you cannot process the same message more than once.

1. Create a subscription.

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

2. Pull messages from the subscription.

```
$ gcloud beta pubsub subscriptions pull messages-subscription-1
Listed 0 items.
```

3. The message you posted earlier will not show up just yet. It's because the message was published prior creation of a subscription. Go back to the frontend application and post another message, and then pull the message again.

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

You should see the message. However, the message will remain in the subscription until it's acknowledged.

4. To pull the message and remove it from subscription from the command line, use auto-acknowledgement.

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

Task 7 - Process Messages in Subscription

You can use `PubSubTemplate` to listen to subscriptions.

1. In a new tab, generate a brand new project from Spring Initializr.

```
$ cd ~
$ curl https://start.spring.io/starter.tgz \
  -d dependencies=cloud-gcp-pubsub \
  -d baseDir=message-processor | tar -xzf -
```

This will generate a new Spring Boot project with Cloud Pub/Sub starter pre-configured.

2. If you are using the Cloud Shell Code Editor, click **File** → **Refresh** to see the newly created directories/files.

3. Open `message-processor/pom.xml` to validate the starter dependency was automatically added.

`message-processor/pom.xml`

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
...
```

4. Write the code to listen to new messages delivered to the topic.

`message-processor/src/main/java/com/example/demo/DemoApplication.java`

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
```

```

import org.springframework.boot.autoconfigure.SpringBootApplication;

// Add imports
import org.springframework.context.annotation.Bean;
import org.springframework.boot.CommandLineRunner;
import org.springframework.cloud.gcp.pubsub.core.*;

@SpringBootApplication
public class DemoApplication {
    @Bean
    public CommandLineRunner cli(PubSubTemplate pubSubTemplate) {
        return (args) -> {
            pubSubTemplate.subscribe("messages-subscription-1",
                (msg, ackConsumer) -> {
                    System.out.println(msg.getData().toStringUtf8());
                    ackConsumer.ack();
                });
        };
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}

```

5. Start message-processor to listen to the topic.

```

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

```

6. Browse to the frontend again, and post a few messages.

7. Validate that the Pub/Sub messages are received in the Message Processor.

```

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

```

Lab 5b - Using Spring Integration for Pub/Sub

Task 1 - Add Spring Integration Core

1. In the Guestbook Frontend, add the Spring Cloud Integration GCP.

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>org.springframework.integration</groupId>
            <artifactId>spring-integration-core</artifactId>
        </dependency>
    </dependencies>
    ...
</project>

```

Task 2 - Create an Outbound Message Gateway

In Spring Integration, you can create a message gateway interface that can abstract away the underlying messaging system used. This way, you can interchange messaging middleware between on-premise applications vs. Cloud-based applications. It also makes it really easy to migrate between messaging middlewares.

1. Create a `OutboundGateway.java` with a single method to send a text message.

`guestbook-frontend/src/main/java/com/example/frontend/OutboundGateway.java`

```

package com.example.frontend;

import org.springframework.integration.annotation.MessagingGateway;

@MessagingGateway(defaultRequestChannel = "messagesOutputChannel")
public interface OutboundGateway {
    void publishMessage(String message);
}

```

Task 3 - Publish Message

This will allow you to use `OutboundGateway` to publish messages. Modify the application to publish the message in `FrontendController.post` method. Whenever someone posts a new Guestbook message, also send it to a messaging system. Notice at this point, the application is unaware of the actual messaging system being used.

1. Add `OutboundGateway` to `FrontendController` and use it to publish a message.

`guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`

```

package com.example.frontend;

...

@Controller
@SessionAttributes("name")

```

```

public class FrontendController {
    @Autowired
    private GuestbookMessagesClient client;

    @Autowired
    private PubSubTemplate pubSubTemplate;

    @Autowired
    private OutboundGateway outboundGateway;

    @Value("${greeting:Hello}")
    private String greeting;

    @GetMapping("/")
    public String index(Model model) {
        ...
    }

    @PostMapping("/post")
    public String post(@RequestParam String name, @RequestParam String
message, Model model) {
        model.addAttribute("name", name);
        if (message != null && !message.trim().isEmpty()) {
            // Post the message to the backend service
            ...

            pubSubTemplate.publish("messages", name + ": " + message);
            outboundGateway.publishMessage(name + ": " + message);
        }
        return "redirect:/";
    }
}

```

Task 4 - Bind Output Channel to Pub/Sub Topic

In the Outbound Gateway, you specified `messagesOutputChannel` as the default request channel. We need to define that channel needs to send the message to the Pub/Sub topic. You can create a new bean for that in `FrontendApplication.java`.

1. Configure a Service Activator to bind the `messagesOutputChannel` to use Cloud Pub/Sub.

`guestbook-frontend/src/main/java/com/example/frontend/FrontendApplication.java`

```

package com.example.frontend;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.hateoas.config.EnableHypermediaSupport;

```

```

import org.springframework.context.annotation.*;
import org.springframework.cloud.gcp.pubsub.core.*;
import org.springframework.cloud.gcp.pubsub.integration.outbound.*;
import org.springframework.integration.annotation.*;
import org.springframework.messaging.*;

// Enable consumption of HATEOS payloads
@EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL)
// Enable Feign Clients
@EnableFeignClients
@SpringBootApplication
public class FrontendApplication {

    public static void main(String[] args) {
        SpringApplication.run(FrontendApplication.class, args);
    }

    @Bean
    @ServiceActivator(inputChannel = "messagesOutputChannel")
    public MessageHandler messageSender(PubSubTemplate pubsubTemplate) {
        return new PubSubMessageHandler(pubsubTemplate, "messages");
    }
}

```

Task 5 - Run it locally

1. Restart the Guestbook Frontend.

```

$ cd ~/guestbook-frontend
$ ./mvnw spring-boot:run -Dspring.profiles.active=cloud

```

2. Open Cloud Shell web preview and post a message.
3. Check for the published messages using either `gcloud` or the Message Processor.

For example, if the Message Processor is still running, check the Message Processor for incoming messages.

Otherwise, use `gcloud` command line to pull the latest message.

```

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

```

Note: Spring Integration for Pub/Sub works for both inbound messages and outbound messages. There is also Pub/Sub support for Spring Cloud Stream to create reactive microservices.

Lab 6 - Uploading and Storing Files

Duration: 30:00

Google Cloud Platform has a bucket-based blob storage called Cloud Storage (GCS). Cloud Storage is designed to store large number and amount of binary data, so that you don't need to manage your own file systems and/or file sharing services. Cloud Storage is across many Google Cloud Platform products whenever you need to store files. E.g., you can store data files on GCS and process it in a managed Hadoop (Dataproc) cluster. You can also import structured data into BigQuery for ad hoc data analytics using standard SQL.

In this lab, we'll add the ability to upload an image associated w/ the message, and the image will be stored in Google Cloud Storage.

Task 1 - Add Google Cloud Storage Starter

1. In the Guestbook Frontend, add the Google Cloud Storage starter.

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-starter-storage</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

Task 2 - Create a Bucket

1. Create a Cloud Storage bucket to store the uploaded file. Bucket names are globally unique. Create a new bucket based on the Project ID.

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
$ gsutil mb gs://${PROJECT_ID}
```

Task 3 - Store Uploaded File

1. Update the homepage to be able to accept a file in the form.

guestbook-frontend/src/main/resources/templates/index.html

```
<html>
<head>
  ...
</head>
```

```

<body>
<nav class="navbar navbar-inverse navbar-fixed-top">
    ...
</nav>

<div class="main container">
    <div class="input">
        <!-- Set form encoding type to multipart form data -->
        <form action="/post" method="post" enctype="multipart/form-data">
            <span>Your name:</span><input type="text" name="name"
th:value="${name}"/>
            <span>Message:</span><input type="text" name="message"/>
            <!-- Add a file input -->
            <span>File:</span>
            <input type="file" name="file" accept=".jpg, image/jpeg"/>
            <input type="submit" value="Post"/>
        </form>
    </div>

    ...

</div>

...

</body>
</html>

```

2. Update the FrontendContollor to accept the file.

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java

```

package com.example.frontend;

import ...;
...

import org.springframework.cloud.gcp.core.GcpProjectIdProvider;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.context.ApplicationContext;
import org.springframework.core.io.Resource;
import org.springframework.core.io.WritableResource;
import org.springframework.util.StreamUtils;
import java.io.*;

@Controller
@SessionAttributes("name")
public class FrontendController {
    ...

    // We need the ApplicationContext in order to create a new Resource.
    @Autowired

```



```

private ApplicationContext context;

// We need to know the Project ID, because it's Cloud Storage bucket name
@Autowired
private GcpProjectIdProvider projectIdProvider;

@GetMapping("/")
public String index(Model model) {
    ...
}

// Capture the file in request parameter
@PostMapping("/post")
public String post(
    @RequestParam(name="file", required=false) MultipartFile file,
    @RequestParam String name,
    @RequestParam String message, Model model)
    throws IOException
{
    model.addAttribute("name", name);

    String filename = null;
    if (file != null && !file.isEmpty()
        && file.getContentType().equals("image/jpeg")) {

        // Bucket ID is our Project ID
        String bucket = "gs://" + projectIdProvider.getProjectId();

        // Generate a random file name
        filename = UUID.randomUUID().toString() + ".jpg";
        WritableResource resource = (WritableResource)
            context.getResource(bucket + "/" + filename);

        // Write the file to Cloud Storage using WritableResource
        try (OutputStream os = resource.getOutputStream()) {
            os.write(file.getBytes());
        }
    }

    if (message != null && !message.trim().isEmpty()) {
        // Post the message to the backend service
        Map<String, String> payload = new HashMap<>();
        payload.put("name", name);
        payload.put("message", message);
        // Store the generated file name in the database
        payload.put("imageUri", filename);
        client.add(payload);
    }
    ...
}

```

```
    }  
    return "redirect:/";  
}  
}
```

Task 4 - Run it locally

1. Restart the Guestbook Frontend.

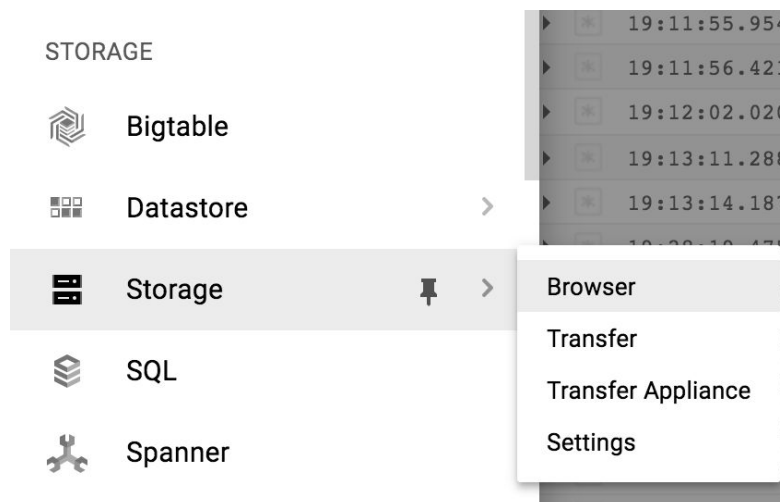
```
$ cd ~/guestbook-frontend  
$ ./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

2. Open Cloud Shell web preview and post a message with a *small* JPEG image. E.g., this one of [New York City](#).

3. Validate that the image was uploaded. From the command line.

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')  
$ gsutil ls gs://${PROJECT_ID}
```

4. From the Google Cloud Platform Console, browse to **Storage** → **Storage** → **Browser**.



6. Navigate to your bucket.

☰

Google Cloud Platform

SpringOne GCP Worksho...

🔍

Storage

Browser

Transfer

Transfer Appliance

Settings

Browser

CREATE BUCKET

REFRESH

DELETE

Filter by prefix...

Buckets

<input type="checkbox"/> Name	Default storage class ?
<input type="checkbox"/> springone17-sfo-1000	Multi-Regional
<input type="checkbox"/> springone17-sfo-1000.appspot.com	Multi-Regional
<input type="checkbox"/> staging.springone17-sfo-1000.appspot.com	Multi-Regional

Where you should see the uploaded file.

Browser

UPLOAD FILES

UPLOAD FOLDER

CREATE FOLDER

REFRESH

Filter by prefix...

Buckets / springone17-sfo-1000

<input type="checkbox"/> Name	Size	Type
<input type="checkbox"/> f53dfc0e-54bb-4791-8e9b-16f21a6728b6.jpg	28.32 KB	application/octet-stream

Task 5 - Serving image from Cloud Storage

1. In `FrontendController`, add a method to retrieve the requested image, and send to the browser.

`guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`

```
package com.example.frontend;

import ...;
...
import org.springframework.http.*;

@Controller
@SessionAttributes("name")
public class FrontendController {
    ...

    @GetMapping("/")
```

```

public String index(Model model) {
    ...
}

// Capture the file in request parameter
@PostMapping("/post")
public String post(...) throws IOException {
    ...
}

// ".+" is necessary to capture URI with filename extension
@GetMapping("/image/{filename:.+}")
public ResponseEntity<Resource> file(@PathVariable String filename) {
    String bucket = "gs://" + projectIdProvider.getProjectId();

    // Use "gs://" URI to construct a Spring Resource object
    Resource image = context.getResource(bucket + "/" + filename);

    // Send it back to the client
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.IMAGE_JPEG);
    return new ResponseEntity<>(image, headers, HttpStatus.OK);
}
}

```

2. Update the homepage so that it'll load the image if present.

guestbook-frontend/src/main/resources/templates/index.html

```

<html>
<head>
    ...
</head>
<body>
<nav class="navbar navbar-inverse navbar-fixed-top">
    ...
</nav>

<div class="main container">
    ...
    <div class="messages">
        <div th:each="message: ${messages}" class="message">
            <span th:text="${message.name}" class="username">Username</span>
            <span th:text="${message.message}" class="message">Message</span>
            
        </div>
    </div>
</div>

</div>

```

```
...  
</body>  
</html>
```

Task 6 - Run it locally

1. Restart the Guestbook Frontend.

```
$ cd ~/guestbook-frontend  
$ ./mvnw spring-boot:run -Dspring.profiles.active=cloud
```

2. Validate that the previous uploaded images are displaying properly.

Lab 7 - Using Google Cloud Platform APIs

Duration: 30:00

In addition to the integration with Spring Boot starters, Google Cloud Platform offers many other APIs that you can use from your application. Google Cloud Platform has ready to use, idiomatic Java client libraries called [google-cloud-java](#). You can consume any of the client libraries even without a Spring Boot starter. Let's try the Vision API to analyze the image you uploaded.

Task 1 - Enable Vision API

1. Enable the Vision API so we can use it to analyze the uploaded images.

```
$ gcloud services enable vision.googleapis.com  
Waiting for async operation operations/... to complete...  
Operation finished successfully. The following command can describe the  
Operation details:  
gcloud services operations describe operations/...
```

Task 2 - Add Google Cloud Vision client library

1. Add the Google Cloud Vision client library to the Guestbook Frontend.

guestbook-frontend/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>  
  ...  
  <dependencies>  
    ...  
    <dependency>  
      <groupId>com.google.cloud</groupId>
```

```
        <artifactId>google-cloud-vision</artifactId>
    </dependency>
</dependencies>
...
</project>
```

Task 3 - Add Scope

Out of the box, the Spring Cloud GCP starters will request permission scopes to use APIs that the starters integrate with. But since we are using a new API that's not integrated with the starter, we'll need to specify the scope. There is a "catch all" scope that can be used to request permission for all basic Google Cloud Platform APIs.

1. Specify the Google Cloud Platform scope in the `application.properties` file.

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

```
spring.cloud.gcp.trace.enabled=false
spring.cloud.gcp.credentials.scopes=https://www.googleapis.com/auth/cloud-platfo
rm
```

Warning: The Google Cloud Platform scope indicates that the application wants to use all of the Google Cloud Platform APIs, However, it's can only use the API if the API is enabled, and also has permission to use it (via the service account roles, or machine credentials).

In a production application, you should always specify the narrowest scopes that the application actually needs to use.

Task 4 - Produce a Vision Client Bean

We'll use Spring to manage the creation of the Vision client library.

1. Add the bean definition to `FrontendApplication.java`.

guestbook-frontend/src/main/java/com/example/frontend/FrontendApplication.java

```
package com.example.frontend;

import org.springframework.boot.SpringApplication;
...
import java.io.IOException;
import com.google.cloud.vision.v1.*;
import com.google.api.gax.core.CredentialsProvider;

@SpringBootApplication
@EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL)
@EnableFeignClients
public class FrontendApplication {
```

```

public static void main(String[] args) {
    SpringApplication.run(FrontendApplication.class, args);
}

...

// This configures the Vision API settings with a credential using the
// the scope we specified in the application.properties.
@Bean
public ImageAnnotatorSettings imageAnnotatorSettings(
    CredentialsProvider credentialsProvider) throws IOException {
    return ImageAnnotatorSettings.newBuilder()
        .setCredentialsProvider(credentialsProvider).build();
}

@Bean
public ImageAnnotatorClient imageAnnotatorClient(
    ImageAnnotatorSettings settings) throws IOException {
    return ImageAnnotatorClient.create(settings);
}
}

```

Note: The client implements `AutoCloseable`, thus its lifecycle is automatically managed by Spring as well.

Task 5 - Analyze Image

Given an image, Google Cloud Vision API can identify objects, landmarks, location of faces and facial expressions, extract text, and whether the image is "safe". In this exercise, we'll analyze the uploaded image, label the objects in the image, and print out the response.

1. Add a method to use the Vision API to analyze an image.

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java

```

package com.example.frontend;

import ...;
...
import com.google.cloud.vision.v1.*;

@Controller
@SessionAttributes("name")
public class FrontendController {
    ...

    @Autowired
    private ImageAnnotatorClient annotatorClient;
}

```

```

    @GetMapping("/")
    public String index(Model model) {
        ...
    }

    private void analyzeImage(String uri) {
        // After the image was written to GCS, analyze it with the GCS URI.
        // Note: It's also possible to analyze an image embedded in the
        // request as a Base64 encoded payload.
        List<AnnotateImageRequest> requests = new ArrayList<>();
        ImageSource imgSrc = ImageSource.newBuilder()
            .setGcsImageUri(uri).build();
        Image img = Image.newBuilder().setSource(imgSrc).build();
        Feature feature = Feature.newBuilder()
            .setType(Feature.Type.LABEL_DETECTION).build();
        AnnotateImageRequest request = AnnotateImageRequest
            .newBuilder()
            .addFeatures(feature)
            .setImage(img)
            .build();

        requests.add(request);
        BatchAnnotateImagesResponse responses =
            annotatorClient.batchAnnotateImages(requests);
        // We send in one image, expecting just one response in batch
        AnnotateImageResponse response = responses.getResponses(0);

        System.out.println(response);
    }

    ...
}

```

2. Call the method after the image was written to the Cloud Storage bucket.

guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java

```

package com.example.frontend;

import ...;
...

@Controller
@SessionAttributes("name")
public class FrontendController {
    ...

    // Capture the file in request parameter
    @PostMapping("/post")
    public String post(...) throws IOException {
        model.addAttribute("name", name);
    }
}

```



```

        String filename = null;
        if (file != null && !file.isEmpty()
            && file.getContentType().equals("image/jpeg")) {
            ...

            try {
                ...
            }

            // After written to GCS, analyze the image.
            analyzeImage(bucket + "/" + filename);
        }

        if (message != null && !message.trim().isEmpty()) {
            ...
        }
        return "redirect:/";
    }

    @GetMapping("/image/{filename:.+}")
    public ResponseEntity<Resource> file(@PathVariable String filename) {
        ...
    }
}

```

Task 6 - Setup a Service Account

For this feature, you'll need to use a service account with the proper permissions to be able to use Cloud Vision API.

1. If you already created a service account in previous lab sections, you can skip to [Task 7](#).
2. If you didn't already create a service account, follow the instructions to create a service account that is of the role `role/editor`. The editor role has a lot of power! In a production environment, you would provision a service account with specific permissions depending on the features your application actually needs.

```

$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
$ gcloud iam service-accounts create guestbook
$ 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

```

Note: This should create a service account credentials file and stored in the `$HOME/service-account.json` file. Treat this file as your own username/password. Do not share this in the public!

Task 7 - Run it locally with Service Account

1. Restart the Guestbook Frontend.

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

2. Post another picture, and you should see the image labels in the log output.

```
label_annotations {
  mid: "/m/01yrx"
  description: "cat"
  score: 0.9918734
  topicality: 0.9918734
}
label_annotations {
  mid: "/m/01l7qd"
  description: "whiskers"
  score: 0.9419696
  topicality: 0.9419696
}
...
```

Lab 8 - Deploy to App Engine

Duration: 30:00

There are many options to deploy your application on Google Cloud Platform. For example, you can deploy the application in a virtual machine, or, containerize your application and deploy into managed Kubernetes cluster. You can also run your favorite PaaS on Google Cloud Platform (e.g., Cloud Foundry, OpenShift, etc).

For this lab, we'll deploy the application into App Engine. App Engine is a platform as a service that scales to 0 when no one is using the service, and scales up automatically.

We need to convert our application (fat WARs) into thin-WAR deployments that App Engine can deploy.

Task 1 - Initialize App Engine

1. Enable App Engine in the project.

```
$ gcloud app create --region=us-central
```

App Engine deployments are regional. I.e., your application may be deployed to multiple zones within a region. Since our CloudSQL instance is in us-central1, we should also also deploy the application into the same region.

Task 2 - Make Guestbook Frontend App Engine Friendly

1. Add the App Engine Plugin to guestbook-frontend's pom.xml.

guestbook-frontend/pom.xml

```
...
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>appengine-maven-plugin</artifactId>
                <version>1.3.1</version>
                <configuration>
                    <version>1</version>
                </configuration>
            </plugin>
        </plugins>
    </build>
...
```

2. Create a WEB-INF directory in Guestbook Frontend.

```
$ mkdir -p ~/guestbook-frontend/src/main/webapp/WEB-INF/
```

3. Add the appengine-web.xml that is needed to deploy to App Engine.

guestbook-frontend/src/main/webapp/WEB-INF/appengine-web.xml

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <service>default</service>
  <version>1</version>
  <threadsafe>true</threadsafe>
  <runtime>java8</runtime>
  <instance-class>B4_1G</instance-class>
  <sessions-enabled>true</sessions-enabled>
  <manual-scaling>
    <instances>2</instances>
  </manual-scaling>
  <system-properties>
```

```
<property name="spring.profiles.active" value="cloud" />
</system-properties>
</appengine-web-app>
```

Note: This configuration uses manual scaling rather than automatic scaling. This is great if you want to have fine control over the number of application instances. However, in a production setting, you may want to use automatic scaling that can adapt dynamically to the load.

Task 3 - Configure Frontend to use the backend URL

Since the frontend application is using Spring Cloud GCP Config starter - it can actually get the backend endpoint URL directly from Google Cloud Platform's Runtime Config.

Configure `messages.endpoint` configuration variable to use the backend URL. There are 2 different ways to do this. Choose only one of the ways:

1. If you were instructed to and finished [Lab 3 - Runtime Configuration](#), then set the backend URL via runtime configuration.

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
$ gcloud beta runtime-config configs variables set messages.endpoint \
  "https://guestbook-service-dot-${PROJECT_ID}.appspot.com/guestbookMessages" \
  --config-name frontend_cloud
```

2. If you did not complete Lab 3, then override the property value in `appengine-web.xml`.

`guestbook-frontend/src/main/webapp/WEB-INF/appengine-web.xml`

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  ...
  <system-properties>
    <property name="spring.profiles.active" value="cloud" />
    <property name="messages.endpoint"
value="https://guestbook-service-dot-PROJECT_ID.appspot.com/guestbookMessages"
/>
  </system-properties>
</appengine-web-app>
```

Task 4 - Deploy to Frontend to App Engine

1. Use Maven to deploy the application.

```
$ cd ~/guestbook-frontend
$ ./mvnw appengine:deploy -DskipTests
```

```
...
[INFO] GCLOUD: Deployed service [default] to [https://PROJECT_ID.appspot.com]
[INFO] GCLOUD:
[INFO] GCLOUD: You can stream logs from the command line by running:
[INFO] GCLOUD:   $ gcloud app logs tail -s default
[INFO] GCLOUD:
[INFO] GCLOUD: To view your application in the web browser run:
[INFO] GCLOUD:   $ gcloud app browse
...
```

2. Find the Frontend URL.

```
$ gcloud app browse
Did not detect your browser. Go to this link to view your app:
https://....appspot.com ← This is your URL!
```

3. Browse to the frontend URL, but it will produce an error.

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sat Jul 14 20:55:36 UTC 2018
There was an unexpected error (type=Internal Server Error, status=500).
status 404 reading GuestbookMessagesClient#getMessages(); content: {"timestamp":1531601736455,"status":404,"error":"Not Found","message":"No message available","path":"/guestbookMessages/"}

This is because the backend isn't deployed yet. We'll fix this in a second.

Task 5 - Make Guestbook Service App Engine Friendly

1. Add the App Engine Plugin to guestbook-service's pom.xml.

guestbook-service/pom.xml

```
...
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>appengine-maven-plugin</artifactId>
                <version>1.3.1</version>
                <configuration>
                    <version>1</version>
                </configuration>
            </plugin>
        </plugins>
    </build>
...
```

2. Create a WEB-INF directory in Guestbook Service.

```
$ mkdir -p ~/guestbook-service/src/main/webapp/WEB-INF/
```

3. Add the appengine-web.xml that is needed to deploy to App Engine.

guestbook-service/src/main/webapp/WEB-INF/appengine-web.xml

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <service>guestbook-service</service>
  <version>1</version>
  <threadsafe>true</threadsafe>
  <runtime>java8</runtime>
  <instance-class>B4_1G</instance-class>
  <manual-scaling>
    <instances>2</instances>
  </manual-scaling>
  <system-properties>
    <property name="spring.profiles.active" value="cloud" />
  </system-properties>
</appengine-web-app>
```

Task 6 - Deploy Backend to App Engine

1. Use Maven to deploy the application to App Engine.

```
$ cd ~/guestbook-service
$ ./mvnw appengine:deploy -DskipTests
...
[INFO] GCLOUD: Deployed service [guestbook-service] to
[https://guestbook-service-dot-PROJECT_ID.appspot.com]
[INFO] GCLOUD:
[INFO] GCLOUD: You can stream logs from the command line by running:
[INFO] GCLOUD:   $ gcloud app logs tail -s guestbook-service
[INFO] GCLOUD:
[INFO] GCLOUD: To view your application in the web browser run:
[INFO] GCLOUD:   $ gcloud app browse -s guestbook-service
...
```

2. Find the deployed backend URL.

```
$ gcloud app browse -s guestbook-service
Did not detect your browser. Go to this link to view your app:
https://guestbook-service-dot-....appspot.com ← This is your URL!
```

3. Navigate to your Guestbook Service's URL, and see the following.

```
{
  "_links" : {
    "guestbookMessages" : {
      "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages{?page,size,sort}",
      "templated" : true
    },
    "profile" : {
      "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/profile"
    }
  }
}
```

4. Follow the `href` links to see all of the messages, e.g.:

<https://guestbook-service-dot-PROJECT.appspot.com/guestbookMessages>, and see the past messages you created (since it's connecting to the same CloudSQL instance.

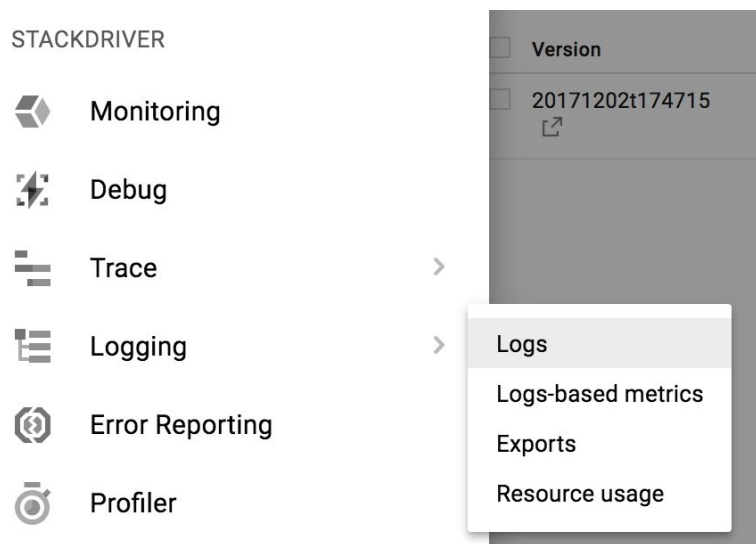
```
{
  "_embedded" : {
    "guestbookMessages" : [ {
      "name" : "Ray",
      "message" : "Hello CloudSQL",
      "imageUri" : null,
      "_links" : {
        "self" : {
          "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages/1"
        },
        "guestbookMessage" : {
          "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages/1"
        }
      }
    }, {
      "name" : "Ray",
      "message" : "Hello",
      "imageUri" : null,
      "_links" : {
        "self" : {
          "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages/2"
        },
        "guestbookMessage" : {
          "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages/2"
        }
      }
    }, {
      "name" : "Ray",
      "message" : "Hi",
      "imageUri" : null,
      "_links" : {
        "self" : {
          "href" : "https://guestbook-service-dot-springone17-sfo-1000.appspot.com/guestbookMessages/3"
        }
      }
    }
  ]
}
```

Lab 9 - DevOps in the Cloud

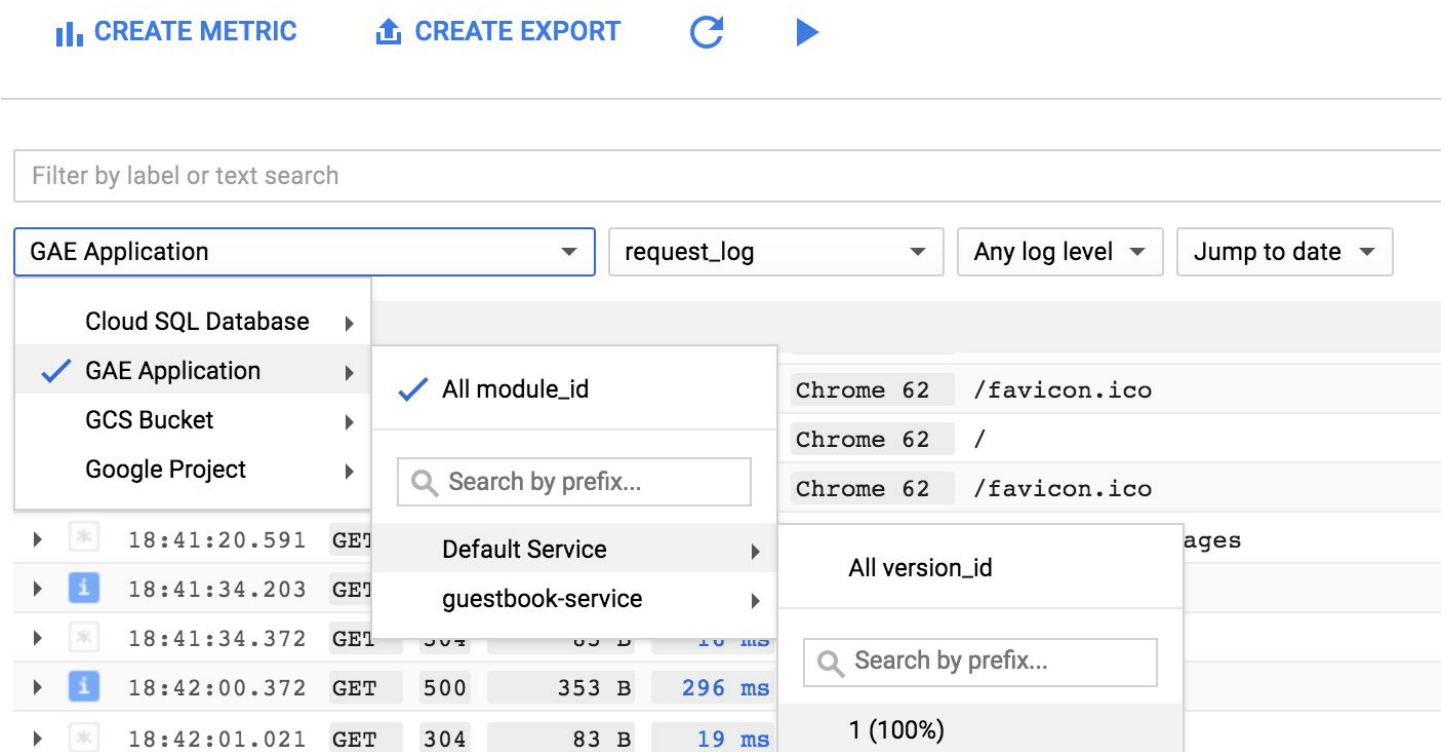
Duration: 1:00:00

Task 1 - Logging

1. Open a new browser tab and navigate to the Google Cloud Platform console.
2. Navigate to **Stackdriver** → **Logs**.



3. In the log drop down, select **GAE Application** → **Default Service** → **1 (100%)**.



4. This is the log from the application. If you output a log message, it'll be grouped by the request. When the application first starts up, the log messages are grouped under `/_ah/start` request.

5. Expand one of the entries and see what's there.


```

18:45:45.656 GET 404 200 B 23.2 s Unknown /_ah/start

0.1.0.3 - - [02/Dec/2017:18:45:45 -0500] "GET /_ah/start HTTP/1.1" 404 200 - "-" "1.1.springone17-sfo-1000.appspot.com" ms=23207 cpu_ms
pm_usd=2.2351e-8 loading_request=1 instance=00c61b117c8691269d9d973e557ade9d0a9ce0993f298b62ff9c4f9c286dedd2fe725cf7dd17d3dc app_engine
=1.9.54 trace_id=ac13100aa4d9a5af3a197b707b4c52b2

{...}

18:45:51.794 javax.servlet.ServletContext log: 2 Spring WebApplicationInitializers detected on classpath

18:45:52.166 [s-springone17-sfo-1000/1.405943748924983130].<stdout>: 23:45:52.162 [Request67EAE920] DEBUG org.springframework.web.
ort.StandardServletEnvironment - Adding PropertySource 'servletConfigInitParams' with lowest search precedence

18:45:52.167 [s-springone17-sfo-1000/1.405943748924983130].<stdout>: 23:45:52.167 [Request67EAE920] DEBUG org.springframework.web.
ort.StandardServletEnvironment - Adding PropertySource 'servletContextInitParams' with lowest search precedence

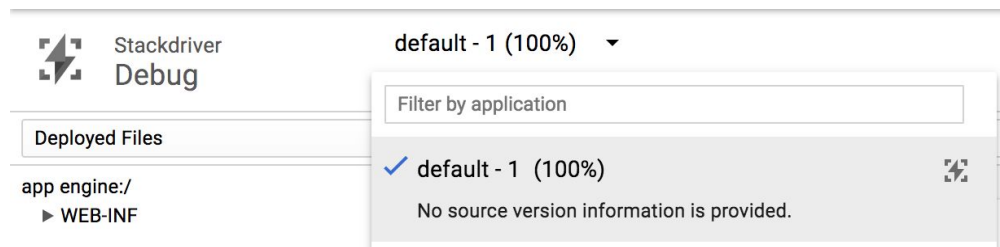
18:45:52.173 [s-springone17-sfo-1000/1.405943748924983130].<stdout>: 23:45:52.173 [Request67EAE920] DEBUG org.springframework.web.
ort.StandardServletEnvironment - Adding PropertySource 'systemProperties' with lowest search precedence

18:45:52.173 [s-springone17-sfo-1000/1.405943748924983130].<stdout>: 23:45:52.173 [Request67EAE920] DEBUG org.springframework.web.
ort.StandardServletEnvironment - Adding PropertySource 'systemEnvironment' with lowest search precedence

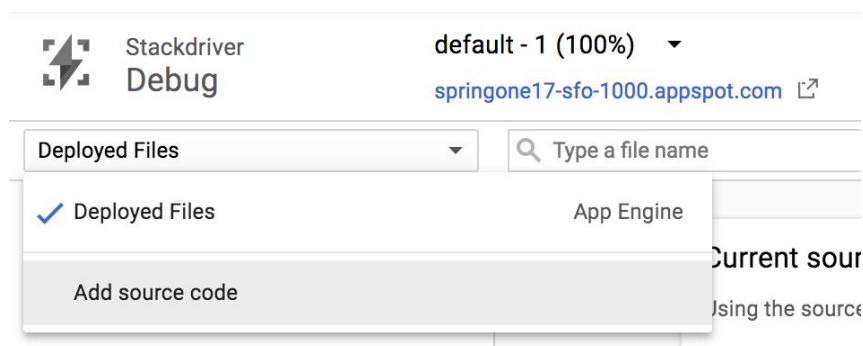
```

Task 2 - Debugging

1. Navigate to **Stackdriver** → **Debug**. On the top, you can see the different App Engine deployments currently running.
2. Select the **default - 1 (100%)** from the drop down.



3. However, there is no source code that we can use for debugging.
4. Navigate to **Deployed Files** → **Add source code**.



5. There are a number of different ways to provide the source code to the Stackdriver Debugger.
6. Scroll down to the end to find the section **Upload a source code capture to Google servers**. Note the command line.

Upload a source code capture to Google servers

Use gcloud to upload a capture of the source code to Google servers.



```
$ gcloud alpha source captures upload --capture-id  
FFEDFB1DB4C4F7051C7CF8EC46DA5D4F19FF85A3 --project springone17-sfo-1000  
LOCAL_PATH
```

7. Enable Source Repository API.

```
$ gcloud services enable sourcerepo.googleapis.com
```

8. Create a Source Repository for Source Capture

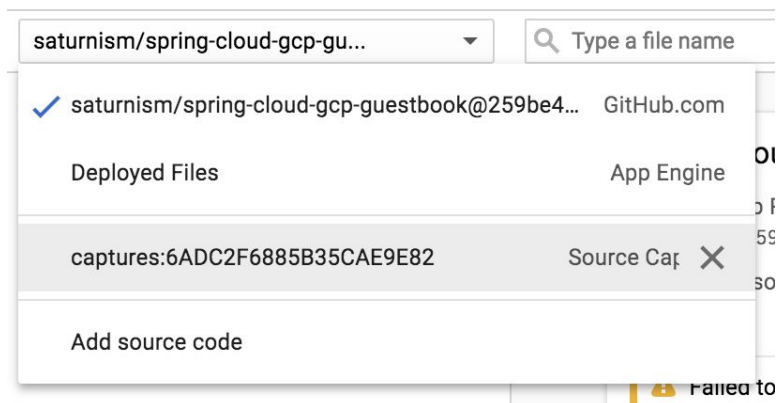
```
$ gcloud source repos create google-source-captures
```

9. Go to Cloud Shell, and use the command line to upload the `guestbook-frontend` source. Please replace the `LOCAL_PATH` with the actual location of the code.

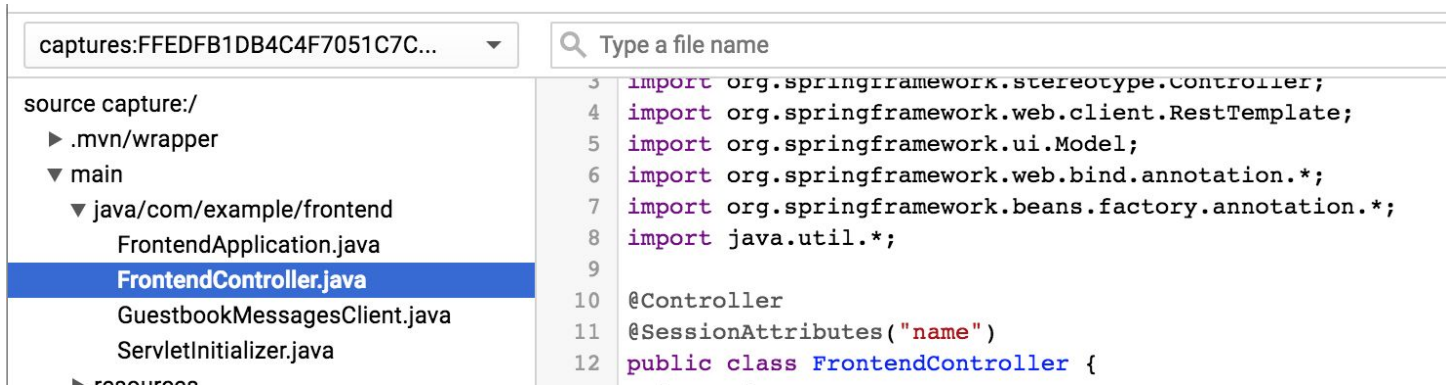
```
$ cd ~/guestbook-frontend  
$ git config --global user.email $(gcloud config get-value core/account)  
$ git config --global user.name "devstar"  
$ gcloud beta debug source upload --project=... --branch=... ./src/
```

10. Go back to the Debugger, console, Refresh the page.

11. Click **Select Source**, and select the newly uploaded Capture.

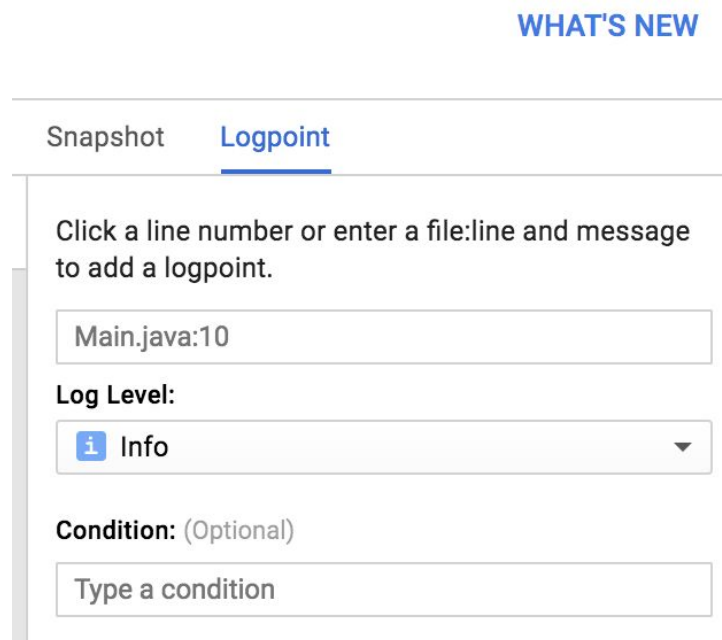


12. Navigate the source and open `FrontendController.java`:

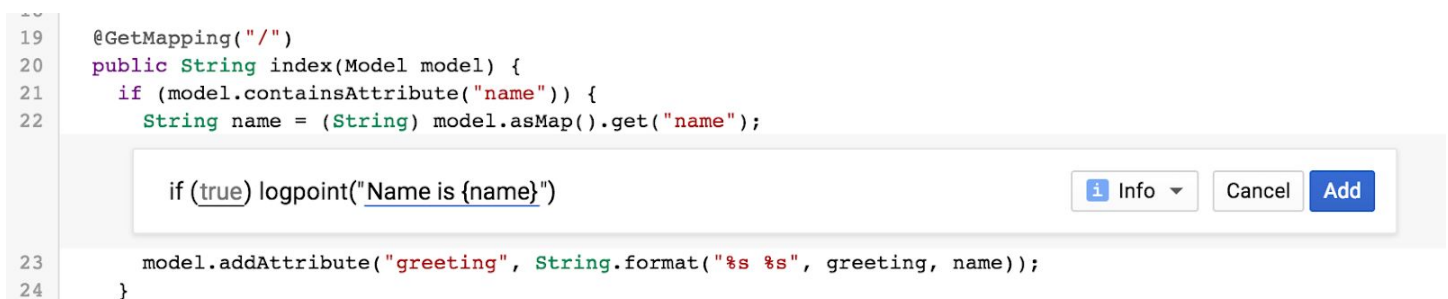


From here, we can do amazing things! For example, add a new log message!

13. On the right hand side, click **Logpoint**.



14. In the source, click on the line number that you want to add a log message, and add the message.



In this example, it'll print the value of the local variable called `name`.

15. Click **Add**. You can add as many log messages as you want.

```

19 @GetMapping("/")
20 public String index(Model model) {
21     if (model.containsKey("name")) {
22         String name = (String) model.asMap().get("name");
23         logpoint("Name is {name}")
24         model.addAttribute("greeting", String.format("%s %s", greeting, name));
25     }
26     logpoint("I'm here!")
27     model.addAttribute("messages", client.getMessage().getContent());
28     return "index";
29 }

```

16. Browse to the site and enter the name/message to trigger the code.

17. Navigate to **Stackdriver** → **Logging**, find the most recent HTTP request to expand it. You should see the new log message.

The screenshot shows the Stackdriver Logging interface. At the top, a log entry is expanded, showing the following details:

- Log Entry:** 19:13:14.141 GET 200 870 B 1.6 s Chrome 62 /;jsessionid=...
- Log Message:** 107.107.59.35 - - [02/Dec/2017:19:13:14 -0500] "GET /;jsessionid=knPTnlh... OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 cpm_usd=9.723e-8 loading_request=0 instance=00c61b117ce2c09f8045b2c286d8...=1.9.54 trace_id=-"
- Log Entry Details:**
 - httpRequest:** {...}
 - insertId:** "5a23419c0009a191b9b0feb3"
 - labels:** {...}
 - logName:** "projects/springone17-sfo-1000/logs/appengine.googleapis.com"
 - operation:** {...}
 - protoPayload:** {...}
 - receiveTimestamp:** "2017-12-03T00:13:16.638835363Z"
 - resource:** {...}
 - severity:** "INFO"
 - timestamp:** "2017-12-03T00:13:14.141769Z"

Below the expanded log entry, two log messages are visible:

- 19:13:14.180 LOGPOINT: Name is Ray (FrontendController.java:23)
- 19:13:14.181 LOGPOINT: I'm here! (FrontendController.java:25)



18. You can also capture the stack in a moment in time. It's almost like stepping through a real debugger, but it does not stop the world.

19. Go back to **Stackdriver** → **Debug**, and in the source view, switch to **Snapshot**.

Snapshot

Logpoint

Click a line number or type file:line to take a snapshot.

Condition: (Optional)

Expressions: (Optional)

20. In the source, simply click on the line number that you want to capture information.



```


18
19  @GetMapping("/")
20  public String index(Model model) {
21      if (model.containsKey("name")) {
22          String name = (String) model.asMap().get("name");
23          logpoint("Name is {name}")
24      }
25      model.addAttribute("greeting", String.format("%s %s", greeting, name));
26      logpoint("I'm here!")
27      model.addAttribute("messages", client.getMessages().getContent());
28      return "index";
29  }

```

21. Browse the page again.

22. As soon as a request flows through the line, the call stack will be captured and you can explore what actually happened.

Variables  

2017-12-02 (19:28:20)
[View request logs](#) 

this

client

greeting

"Hi!"

model

Call Stack

com.example.frontend.FrontendController.index

FrontendController.java:25

sun.reflect.NativeMethodAccessorImpl.invoke0

NativeMethodAccessorImpl.java:-1

sun.reflect.NativeMethodAccessorImpl.invoke

NativeMethodAccessorImpl.java:62

sun.reflect.DelegatingMethodAccessorImpl.invoke

DelegatingMethodAccessorImpl.java:43

java.lang.reflect.Method.invoke

Method.java:498

org.springframework.web.method.support.InvocableHandlerMethod.doInvoke

InvocableHandlerMethod.java:205

org.springframework.web.method.support.InvocableHandlerMethod.invoke

InvocableHandlerMethod.java:133

org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invoke

ServletInvocableHandlerMethod.java:97

23. You can add both **Logpoint** and **Snapshot** with conditionals, so that you look at only certain requests based on variables that are in scope (e.g., session ID).

Note: Cloud Debugger works with different languages, and also outside of App Engine. You can also debug your application in the same way when you deploy your application on-premise, in a VM, or in containers.

Task 3 - Monitoring

1. Navigate to **Stackdriver** → **Monitoring**
2. After selecting your currently logged in account, continue through the wizard to setup Stackdriver Monitoring.
3. Select **Create a new Stackdriver Account**, click **Continue**.
4. Click **Create Account**.
5. When prompted **Add Google Cloud Platform projects to monitor**, do not select any additional projects, click **Continue**.
6. When prompted **Monitor AWS accounts**, click **Skip AWS Setup**.

7. When prompted **Install the Stackdriver Agents**, click **Continue**.
8. When prompted **Get Reports by Email**, select **No reports**, and click **Continue**.
9. Finally, click **Launch monitoring**.

Out of the box, Stackdriver automatically discovers your managed services and ingests the metrics. From here you can customize dashboards, set up alerts, etc.

11. Navigate to **Resources** → **GCP** → **App Engine**.

Lab 10 - Cloud Spanner

Duration: 45:00

When performance and transactions are critical to your application, Cloud Spanner is the relational database service to use. It delivers global transactional consistency that can scale to massive performance levels. Let's see how we can update our application to use the Spring Cloud GCP starter for Cloud Spanner.

Task 1 - Enable Cloud Spanner API

1. Enable the Spanner API.

```
$ gcloud services enable spanner.googleapis.com
```

Task 2 - Create a new Cloud Spanner instance

1. Provision a new Cloud Spanner instance.

```
$ gcloud spanner instances create guestbook --config=regional-us-central1 \
  --nodes=1 --description="Guestbook messages"
Creating instance...done.
```

2. Create a new `messages` database within the Cloud Spanner instance.

```
$ gcloud spanner databases create messages --instance=guestbook
```

3. Confirm the database exists by listing the new instance's databases.

```
$ gcloud spanner databases list --instance=guestbook
NAME          STATE
messages     READY
```

4. Create a new table in the Cloud Spanner database by creating a file containing the DDL statement to be run and then run it. In the `guestbook-service` folder create the folder `db`.

```
$ cd ~/guestbook-service
$ mkdir db
```

5. Create the `spanner.ddl` file containing the DDL statement:

`guestbook-service/db/spanner.ddl`

```
CREATE TABLE guestbook_message (
  id STRING(36) NOT NULL,
  name STRING(255) NOT NULL,
  image_uri STRING(255),
  message STRING(255)
) PRIMARY KEY (id);
```

6. Run the DDL command using `gcloud` to create the table.

```
$ gcloud spanner databases ddl update messages \
  --instance=guestbook --ddl="$(cat db/spanner.ddl)"
```

Task 3 - Add Spring Cloud GCP Cloud Spanner Starter

1. Update the Guestbook Service's `pom.xml` file with a newer version of the Spring Cloud GCP and the Cloud Spanner Starter dependency.

`guestbook-service/pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

  ...

  <dependencies>
    ...
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-starter-data-spanner</artifactId>
    </dependency>
  </dependencies>

  ...
</project>
```


Task 4 - Configure the Cloud Profile to use Cloud Spanner

1. Add Spanner Instance and Database configuration to `application-cloud.properties`, and delete the Cloud SQL configurations.

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

```
spring.cloud.gcp.spanner.instance-id=guestbook
spring.cloud.gcp.spanner.database=messages

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

...
```

Task 5 - Update the backend to use Cloud Spanner

We can use the `@Table` annotation to map a Java class to a Cloud Spanner table, and we can use the `@Column` annotation to map properties to table columns.

In the next code snippet, we will use the `@Table` annotation to map to the `guestbook_message` table that was created when you ran the DDL statement with `gcloud` above.

The `id` property is specified as the primary key. In the class constructor, the `id` property gets auto populated with a random UUID. UUIDv4 is the recommended ID format rather than monotonically increasing ID. This helps Spanner to avoid creating hotspots when Spanner automatically shards the data.

The other class properties included match the table's schema in the DDL statement, except for `imageUri` which uses the `@Column` annotation to map the actual table column name `image_uri` to the property name `imageUri`.

1. Replace `GuestbookMessage.java` to use the Spanner Annotations.

`guestbook-service/src/main/java/com/example/guestbook/GuestbookMessage.java`

```
package com.example.guestbook;

import lombok.*;
import org.springframework.cloud.gcp.data.spanner.core.mapping.*;
import org.springframework.data.annotation.Id;

@Data
@Table(name = "guestbook_message")
public class GuestbookMessage {
    @PrimaryKey
    @Id
    private String id;
```

```

    private String name;

    private String message;

    @Column(name = "image_uri")
    private String imageUrl;

    public GuestbookMessage() {
        this.id = java.util.UUID.randomUUID().toString();
    }
}

```

Task 6 - Add a Method to Find Messages by Name

Spring Data Spanner implements a number of commonly used Spring Data patterns, such as creating simple methods that can automatically get translated to corresponding SQL queries.

For example, a simple method signature `List<GuestbookMessage> findByName(String name);` The Spring framework enables the ability to query our Cloud Spanner table with the SQL query `SELECT * FROM guestbook_message WHERE name = ?`.

1. Update the `GuestbookMessageRepository.java` file so that it uses `String` as the ID type.

`guestbook-service/src/main/java/com/example/guestbook/GuestbookMessageRepository.java`

```

package com.example.guestbook;

import java.util.List;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface GuestbookMessageRepository extends
    PagingAndSortingRepository<GuestbookMessage, String> {
    ...
}

```

2. Update `GuestbookMessageRepository.java` to include the `findByName()` method.

`guestbook-service/src/main/java/com/example/guestbook/GuestbookMessageRepository.java`

```

package com.example.guestbook;

import java.util.List;

import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

```

```
@RepositoryRestResource
public interface GuestbookMessageRepository extends
    PagingAndSortingRepository<GuestbookMessage, String> {

    List<GuestbookMessage> findByName(String name);

}
```

Task 7 - Run it in Cloud Shell

1. Make sure you are in `guestbook-service` directory, and/or kill the existing `guestbook-service` application and do this in that Cloud Shell tab.

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

2. Launch the Guestbook Service with the `cloud` profile.

```
$ ./mvnw spring-boot:run -Dserver.port=8081 -Dspring.profiles.active=cloud \
-Dspring.cloud.gcp.credentials.location=file:/// $HOME/service-account.json
```

3. During the application startup, you should see the Cloud Spanner `SpannerRepositoryFactoryBean` displayed in the logs.

```
...
[org.springframework.cloud.gcp.data.spanner.repository.support.SpannerRepositoryFactoryBean];
...
```

4. In a new Cloud Shell tab, POST a message using `curl`.

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

5. You can also list all the messages.

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

6. Try out the custom `findByName()` search you added above.

```
$ curl http://localhost:8081/guestbookMessages/search/findByName?name=Ray
```

7. You can also use the `gcloud spanner` command to validate with a SQL query.

```
$ gcloud spanner databases execute-sql messages --instance=guestbook \
  --sql="SELECT * FROM guestbook_message WHERE name = 'Ray'"
```

8. In Google Cloud Platform console, browse to **Spanner** → **Guestbook messages** → **messages** → **guestbook_message** → **data** to see the entry.

guestbook_message

Schema Indexes Data

[Insert](#) [Edit](#) [Delete](#)


 Filter by primary key

<input type="checkbox"/> id 	image_uri	message
<input type="checkbox"/> 5b26f86c-909e-4a73-935e-c62b2b4a5b9b	<NULL>	Hello Cloud Spanner

9. Click **Query**, then **Run Query** with the default select query.

Query database: messages

```
1 SELECT * FROM guestbook_message LIMIT 100
```

[Run query](#)  [Clear query](#) [SQL query help](#)

Schema Results table Explanation

Query complete (2.71ms elapsed)

id	name	image_uri	message
5b26f86c-909e-4a73-935e-c62b2b4a5b9b	Ray		Hello Cloud Spanner

Task 8 - Deploy Updated Backend to App Engine

1. Redeploy Guestbook Service to App Engine.

```
$ cd ~/guestbook-service
$ ./mvnw clean appengine:deploy -DskipTests
```

```
...
[INFO] GCLOUD: Deployed service [guestbook-service] to
[https://guestbook-service-dot-PROJECT_ID.appspot.com]
[INFO] GCLOUD:
[INFO] GCLOUD: You can stream logs from the command line by running:
[INFO] GCLOUD:   $ gcloud app logs tail -s guestbook-service
[INFO] GCLOUD:
[INFO] GCLOUD: To view your application in the web browser run:
[INFO] GCLOUD:   $ gcloud app browse -s guestbook-service
...
```

2. Browse and test it out.

```
$ gcloud app browse -s guestbook-service
Did not detect your browser. Go to this link to view your app:
https://guestbook-service-dot-....appspot.com ← This is your URL!
```

3. Browse the Guestbook Frontend too.

```
$ gcloud app browse -s default
Did not detect your browser. Go to this link to view your app:
https://guestbook-service-dot-....appspot.com ← This is your URL!
```

Bonus Lab 11 - Kubernetes

Task 1 - Containerize the Applications

1. Enable Container Registry API.

```
$ gcloud services enable containerregistry.googleapis.com
```

2. Find the Project ID for the current project. You'll need to use it to replace PROJECT_ID with the actual Project ID you retrieve here.

```
$ gcloud config list --format 'value(core.project)'
```

3. Remove Provided scope from the Tomcat dependency, and add Jib plugin to Frontend's pom.xml.

guestbook-frontend/pom.xml

```
...
    <dependencies>
        ...
```

```

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        ...
    </dependencies>
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>0.9.6</version>
                <configuration>
                    <to>
                        <image>gcr.io/PROJECT_ID/guestbook-frontend</image>
                    </to>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...

```

4. Build the Frontend Container.

```

$ cd ~/guestbook-frontend
$ ./mvnw clean compile jib:build
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:frontend
>-----
[INFO] Building frontend 0.0.1-SNAPSHOT
[INFO] -----[ war
]-----
[INFO]
[INFO] --- jib-maven-plugin:0.9.6:build (default-cli) @ frontend ---
[WARNING] Base image 'gcr.io/distroless/java' does not use a specific image
digest - build may not be reproducible
[INFO]
[INFO] Containerizing application to gcr.io/.../...
[INFO]
[INFO] Retrieving registry credentials for gcr.io...
[INFO] Getting base image gcr.io/distroless/java...
[INFO] Building dependencies layer...
[INFO] Building resources layer...
[INFO] Building classes layer...
[INFO] Finalizing...
[INFO]
[INFO] Container entrypoint set to [java, -cp,
/app/libs/*:/app/resources:/app/classes/,

```

```

com.example.frontend.FrontendApplication]
[INFO]
[INFO] Built and pushed image as
gcr.io/next18-bootcamp-test/spring-cloud-gcp-guestbook-frontend
[INFO]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 43.730 s
[INFO] Finished at: 2018-07-16T16:07:34-04:00
[INFO]
-----

```

5. Add Jib plugin to Backend's pom.xml.

guestbook-service/pom.xml

```

...
    <dependencies>
        ...
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        ...
    </dependencies>

    <build>
        <plugins>
            ...

            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>0.9.6</version>
                <configuration>
                    <to>
                        <image>gcr.io/PROJECT_ID/guestbook-service</image>
                    </to>
                </configuration>
            </plugin>
        </plugins>
    </build>
...

```

6. Build the Backend Container.

```

$ cd ~/guestbook-service
$ ./mvnw clean package jib:build
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:frontend
>-----
[INFO] Building frontend 0.0.1-SNAPSHOT
[INFO] -----[ war
]-----
[INFO]
[INFO] --- jib-maven-plugin:0.9.6:build (default-cli) @ frontend ---
[WARNING] Base image 'gcr.io/distroless/java' does not use a specific image
digest - build may not be reproducible
[INFO]
[INFO] Containerizing application to gcr.io/.../...
[INFO]
[INFO] Retrieving registry credentials for gcr.io...
[INFO] Getting base image gcr.io/distroless/java...
[INFO] Building dependencies layer...
[INFO] Building resources layer...
[INFO] Building classes layer...
[INFO] Finalizing...
[INFO]
...
[INFO]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 43.730 s
[INFO] Finished at: 2018-07-16T16:07:34-04:00
[INFO]
-----

```

Task 2 - Create a Kubernetes Cluster

1. Enable Kubernetes Engine API.

```
$ gcloud services enable container.googleapis.com
```

2. Create a Kubernetes Cluster that has Stackdriver Monitoring enabled.

```
$ gcloud beta container clusters create guestbook-cluster \
  --cluster-version=1.10 \
  --region=us-central1 \
  --num-nodes=2 \
  --machine-type=n1-standard-2 \
  --enable-autorepair \

```



```
--enable-stackdriver-kubernetes
```

3. Validate that the Kubernetes cluster was created successfully by checking the server version.

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"10", GitVersion:"v1.10.5",
GitCommit:"32ac1c9073b132b8ba18aa830f46b77dcceb0723", GitTreeState:"clean",
BuildDate:"2018-06-22T05:40:33Z", GoVersion:"go1.9.7", Compiler:"gc",
Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"10+",
GitVersion:"v1.10.5-gke.0",
GitCommit:"f4c74e18e57148052c59cc0467bb7e99dcc46399", GitTreeState:"clean",
BuildDate:"2018-06-21T14:11:26Z", GoVersion:"go1.9.3b4", Compiler:"gc",
Platform:"linux/amd64"}
```

Task 3 - Storing Service Account

The service account you generated earlier needs to be stored in Kubernetes as a Secret so that it's accessible from the containers.

1. Create the Secret from the service account.

```
$ kubectl create secret generic guestbook-service-account \
--from-file=$HOME/service-account.json
```

2. Validate that the service account is stored.

```
$ kubectl describe secret guestbook-service-account
Name:          guestbook-service-account
Namespace:     default
Labels:        <none>
Annotations:   <none>

Type: Opaque

Data
====
service-account.json:  ... bytes
```

Task 4 - Deploying Containers

1. Copy the pre-created Kubernetes deployments file to the home directory.

```
$ cd ~/
```

```
$ cp -a ~/spring-cloud-gcp-guestbook/11-kubernetes/kubernetes ~/kubernetes
```

2. Update the Guestbook Frontend Kubernetes Deployment files to use the image you created.

kubernetes/guestbook-frontend-deployment.yaml

```
...
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: guestbook-frontend
    name: guestbook-frontend
spec:
  replicas: 2
  selector:
    ...
  template:
    ...
    spec:
      ...
      containers:
      - name: guestbook-frontend
        image: saturnism/spring-gcp-guestbook-frontend:latest
        image: gcr.io/PROJECT_ID/guestbook-frontend
      ...
```

3. Update the Guestbook Frontend Kubernetes Deployment files to use the image you created.

kubernetes/guestbook-service-deployment.yaml

```
...
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: guestbook-service
    name: guestbook-service
spec:
  replicas: 2
  selector:
    ...
  template:
    ...
    spec:
      ...
      containers:
      - name: guestbook-frontend
        image: saturnism/spring-gcp-guestbook-frontend:latest
```

```
image: gcr.io/PROJECT_ID/guestbook-service
...
```

4. Deploy the updated Kubernetes deployments

```
$ kubectl apply -f ~/kubernetes/
```

5. Guestbook Frontend is configured to deploy an external Load Balancer. It'll generate an external IP address that does L4 Load Balancing to your backend. Check and wait until the external IP is populated.

```
$ kubectl get svc guestbook-frontend
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
guestbook-frontend	LoadBalancer	...	23.251.156.216

6. Browse to the application on the URL of, `http://EXTERNAL_IP:8080`.

Bonus Lab 12 - Kubernetes Monitoring

Task 1 - Stackdriver Kubernetes Monitoring

In previous lab, you created a Kubernetes Cluster with Stackdriver Kubernetes Monitoring support. That means we can monitor the health of the Kubernetes cluster.

1. Browse to Google Cloud Platform console.
2. Click **Stackdriver** → **Monitoring** to open Stackdriver Monitoring console.
3. Click **Resources** → **Kubernetes** to view Kubernetes Monitoring dashboard.

Task 2 - Enable Prometheus Monitoring

Traditionally, Java applications are monitored via JMX metrics, which may have metrics on thread count, heap usage, etc. In the Cloud Native world where you monitor more than just Java stack, you need to use more generic metrics formats, such as Prometheus.

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

1. Install Prometheus Agent RBAC.

```
$ kubectl apply -f \
https://storage.googleapis.com/stackdriver-prometheus-documentation/rbac-setup.y
ml \
```

```
--as=admin --as-group=system:masters
```

2. Install Prometheus Agent.

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
$ curl -s \
https://storage.googleapis.com/stackdriver-prometheus-documentation/prometheus-s
ervice.yml | \
  sed -e "s/(\s*_kubernetes_cluster_name:*\) .*/\1 'guestbook-cluster'/g" | \
  sed -e "s/(\s*_kubernetes_location:*\) .*/\1 'us-central1'/g" | \
  sed -e "s/(\s*_stackdriver_project_id:*\) .*/\1 '${PROJECT_ID}'/g" | \
  kubectl apply -f -
```

This will install Prometheus agent in the `stackdriver` namespace.

3. Validate that the Prometheus agent is running.

```
$ kubectl get pods -n stackdriver
```

Task 3 - Expose Prometheus Metrics from the Spring Boot applications

Luckily, Spring Boot can expose metrics information via Spring Boot Actuator, and with the combination of Micrometer, it can expose all the metrics with the Prometheus format. It is easy to add Prometheus support.

If you are not using Spring Boot, you can expose JMX metrics via Prometheus by using a [Prometheus JMX Exporter agent](#).

1. Add the Actuator Starter and Micrometer dependencies to Guestbook Frontend.

guestbook-frontend/pom.xml

```
...
<dependencies>
  ...
  <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>
</dependencies>
...
```

2. Configure the Actuator to expose metrics on port 8081.

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

```
...  
  
management.server.port=8081  
management.endpoints.web.exposure.include=*
```

3. Rebuild the container.

```
$ cd ~/guestbook-frontend  
$ ./mvnw clean package jib:build
```

4. Update the Kubernetes deployment to specify the Prometheus metrics endpoint. With this configuration, Spring Boot Actuator will expose the Prometheus metrics on port 8081, under the path of /actuator/prometheus.

Add Prometheus annotations to the deployment.spec.template.metadata.annotation section.

kubernetes/guestbook-frontend.yaml

```
...  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  labels:  
    app: guestbook-frontend  
    name: guestbook-frontend  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: guestbook-frontend  
  template:  
    metadata:  
      labels:  
        app: guestbook-frontend  
      annotations:  
        prometheus.io/scrape: 'true'  
        prometheus.io/path: '/actuator/prometheus'  
        prometheus.io/port: '8081'  
    ...
```

5. Re-deploy the configuration.

```
$ cd ~/kubernetes  
$ kubectl apply -f guestbook-frontend-deployment.yaml
```

6. The Prometheus Metrics will take a couple of minutes to scrape and be displayed in Stackdriver Monitoring. In the mean time, `curl` the Prometheus endpoint to make sure it exists. Find the pod names first.

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

NAME	READY	STATUS	RESTARTS	AGE
guestbook-frontend-8567fdc8c8-c68vk	1/1	Running	0	5m
guestbook-frontend-8567fdc8c8-gvcf5	1/1	Running	0	5m

7. Establish a port forward to one of the Guestbook Frontend pod.

```
$ kubectl port-forward guestbook-frontend-... 8081:8081
```

8. In a new tab, `curl` the Prometheus endpoint.

```
$ curl http://localhost:8081/actuator/prometheus
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is
committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="nonheap",id="Code Cache",} 1.8284544E7
jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 6.6281472E7
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",}
8609792.0
jvm_memory_committed_bytes{area="heap",id="PS Eden Space",} 6.01358336E8
jvm_memory_committed_bytes{area="heap",id="PS Survivor Space",} 2.2020096E7
jvm_memory_committed_bytes{area="heap",id="PS Old Gen",} 1.1010048E8
# HELP tomcat_global_sent_bytes_total
...
```

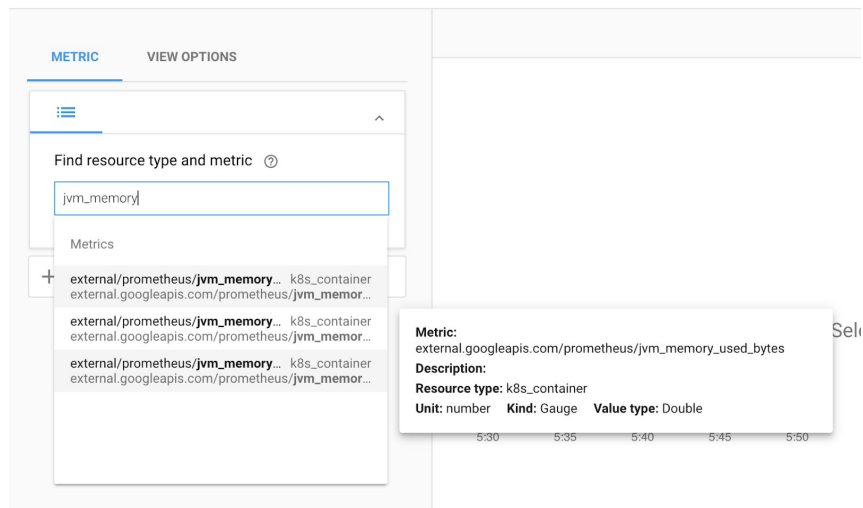
All of these metrics are going to be available inside of Stackdriver Monitoring for visualization, building dashboards, and also setting up alerts. Some of these metrics (e.g., `jvm_memory_committed_bytes`) has multiple dimensions (e.g., `area` and `id`). These dimensions will also be filterable/groupable within Stackdriver Monitoring too.

7. Navigate the Stackdriver Monitoring console. Refresh the page if you already have the page open.

8. Click **Resources** → **Metrics Explorer**.

9. In the Metrics Explorer, search for `jvm_memory` to find some metrics collected by the Prometheus Agent from Spring Boot application.

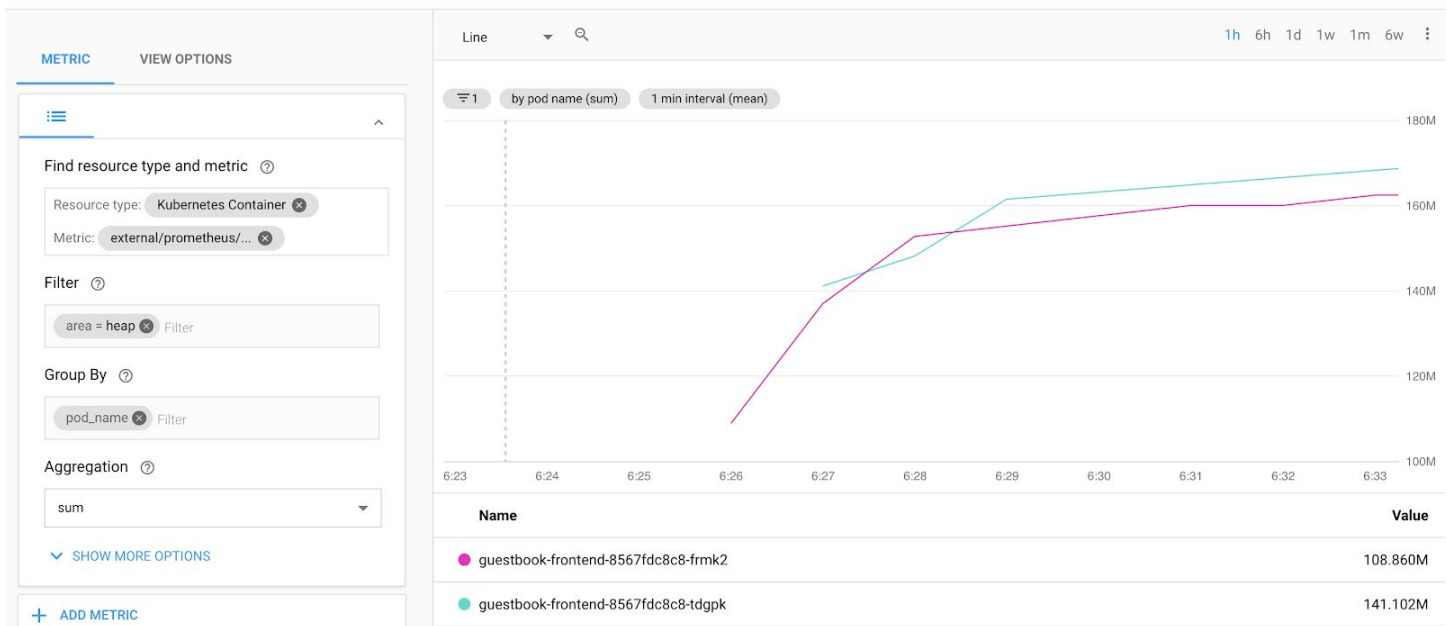
Metrics Explorer



10. Select `jvm_memory_used` to plot the metrics.

12. There are multiple dimensions to the JVM Memory, e.g. Heap vs Non-Heap, and Eden Space vs Metaspace, etc. In **Filter by**, filter by `area=Heap`. In **Group by**, group by `pod_name`. In **Aggregation**, select **Sum**.

Metrics Explorer



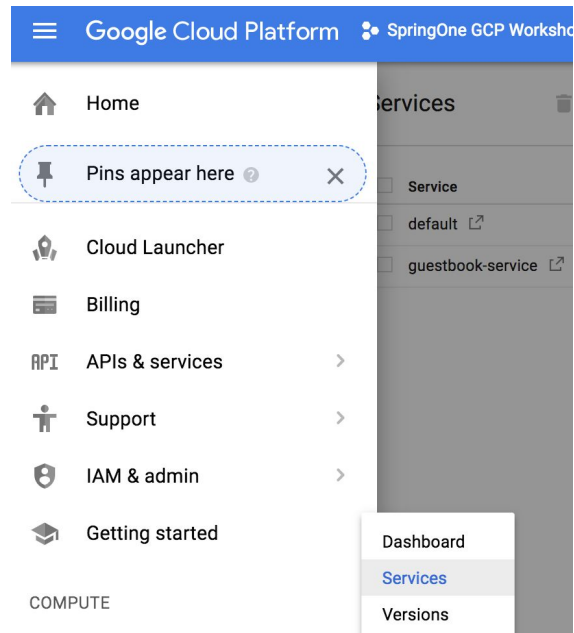
This should build a graph of current Heap usage of the frontend application.

Lab 13 - Clean Up

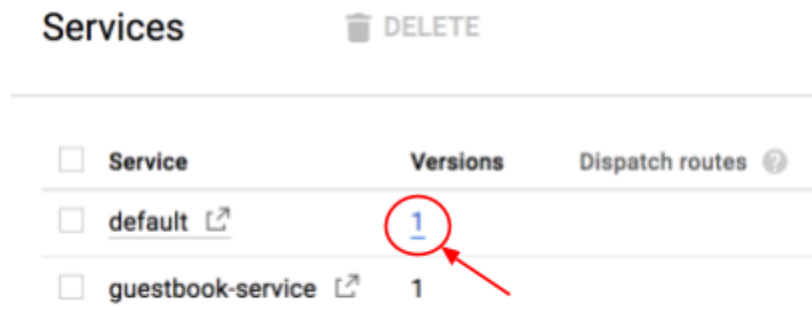
Before you leave, please make sure to clean up the project and remove public facing deployments such as the Guestbook Frontend and service.

Stop App Engine Apps

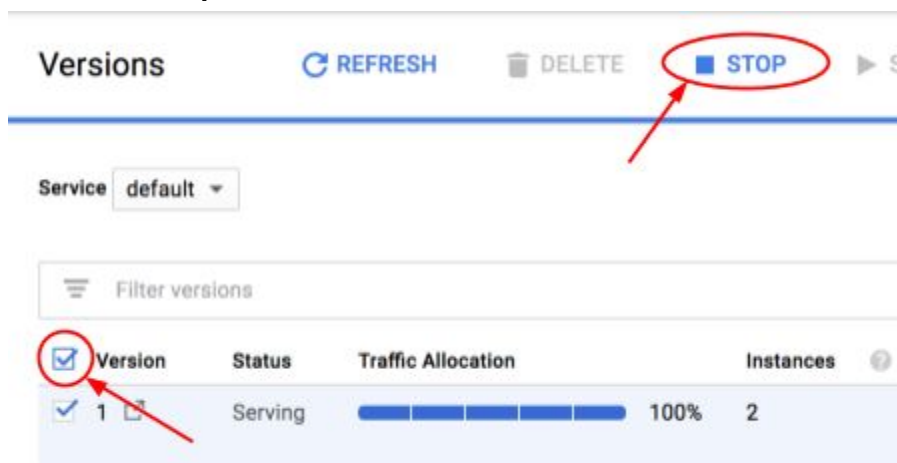
1. Navigate to **Compute** → **App Engine** → **Services**



2. For each of the App Engine services, click the number under **Version**.



3. Select all versions, then click **Stop**.



4. Delete all Cloud SQL instances.
5. Delete all Spanner instances.
6. Delete all Kubernetes Engine clusters.