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

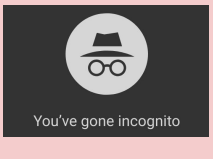
Lab 0 - Initial setup

Duration: 20:00

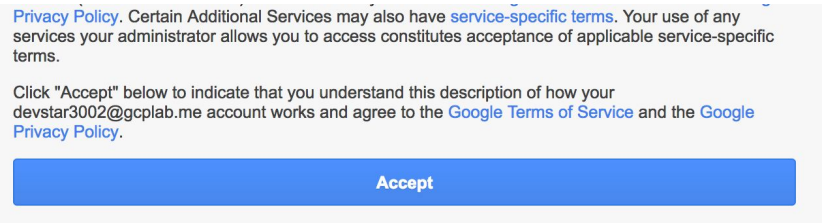
Task 1 - Logging In

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

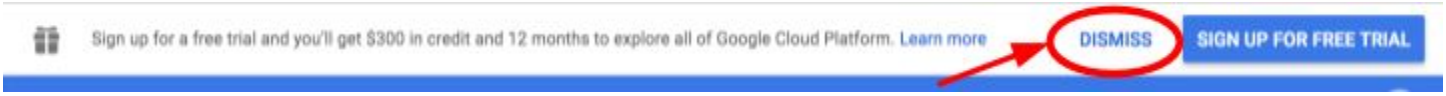
Important: To avoid conflicts with your personal account, please open a new incognito window for the rest of this lab.



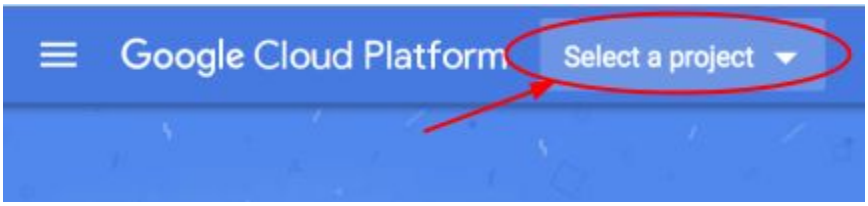
2. Sign in to the Google Cloud Platform Console: https://console.cloud.google.com/ with the provided credentials. In **Welcome to your new account** dialog, click **Accept**.



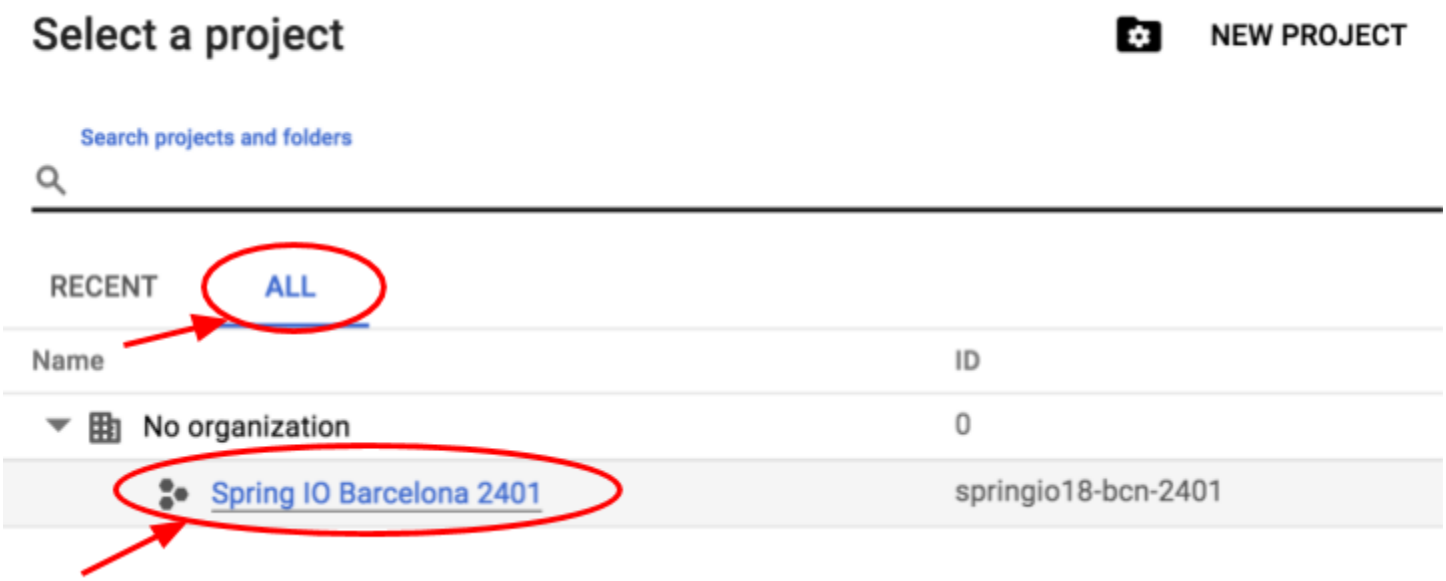
3. If you see a top bar with **Sign Up for Free Trial** - DO NOT SIGN UP FOR THE FREE TRIAL. Click **Dismiss** since you'll be using a pre-provisioned lab account. If you are doing this on your own account, then you may want the free trial.



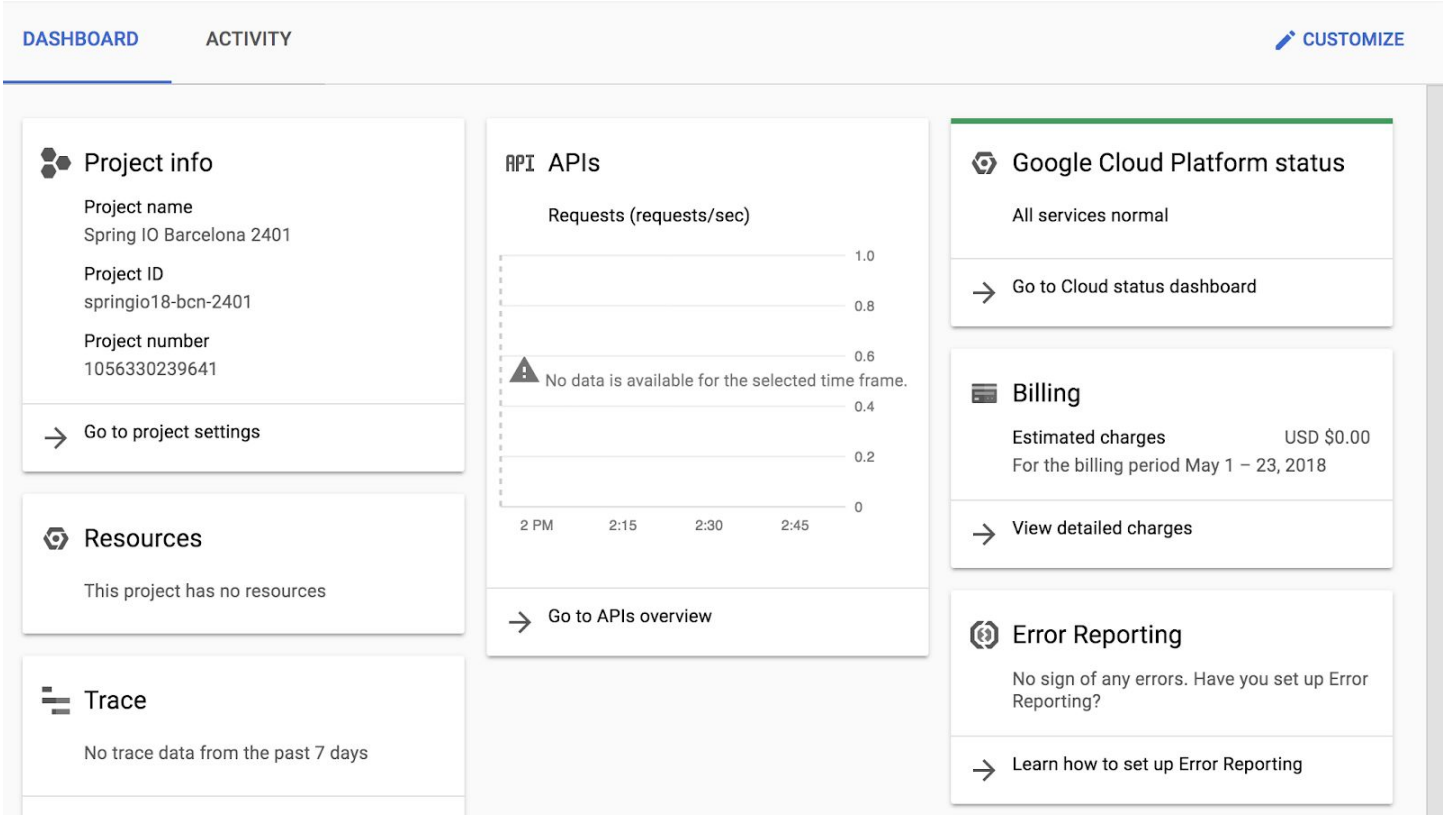
4. Click **Select a project.**



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



You should see the Project Dashboard:

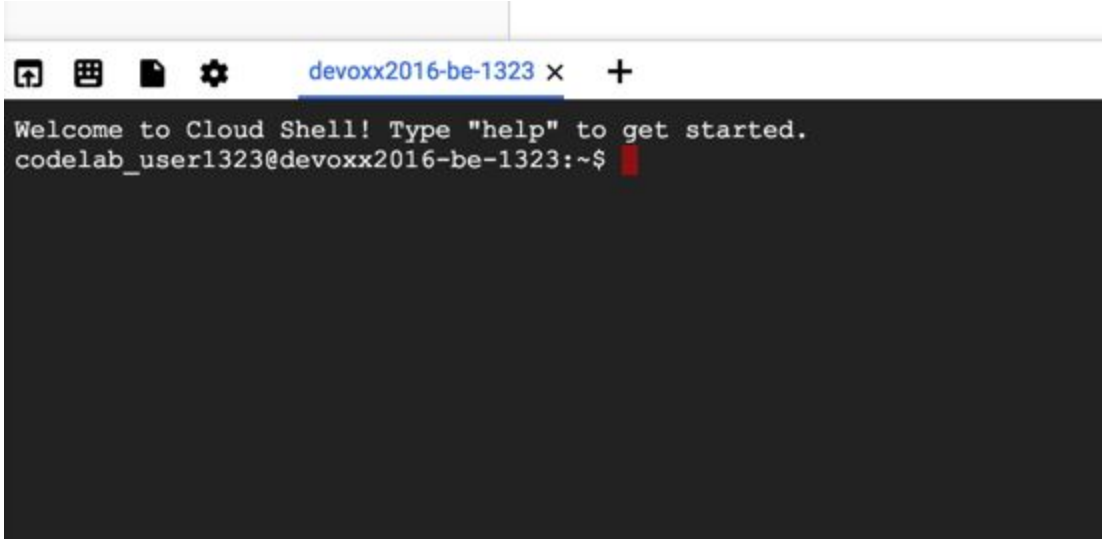


Task 2 - Cloud Shell

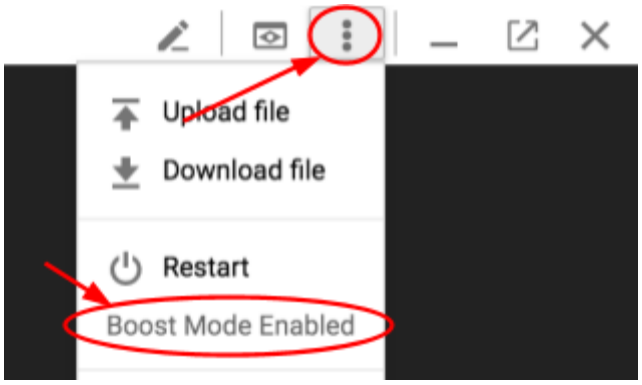
You will do most of the work from the Google Cloud Shell, a command line environment running in the Cloud. This Debian-based virtual machine is loaded with all the development tools you'll need (`docker`, `gcloud`, `kubectl` and others) and offers a persistent 5GB home directory. Open the Google Cloud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

Note: The Backend was bootstrapped using Spring Boot Initializr with Spring Web, Lombok, Spring Data JPA, Rest Repositories, and GCP Support.

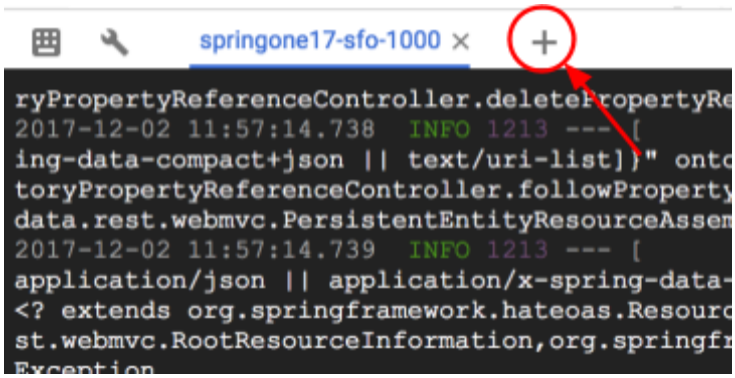
Task 2 - Run Backend

1. Test it out.

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

This will start the backend on port 8081. It is configured in the `src/main/resources/application.properties` file.

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 Frontend section.

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

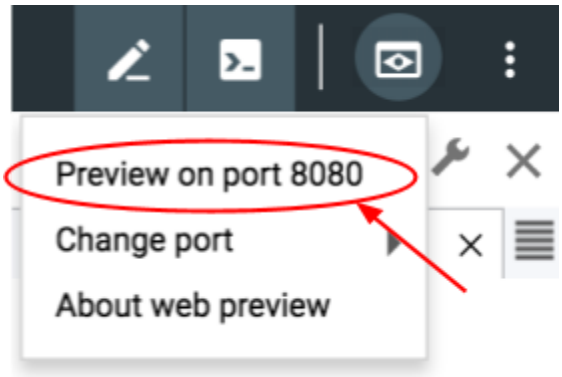
Note: The Frontend was bootstrapped using Spring Boot Initializr with Web, Lombok, Thymeleaf, OpenFeign, and GCP Support.

Taks 4 - Run Frontend

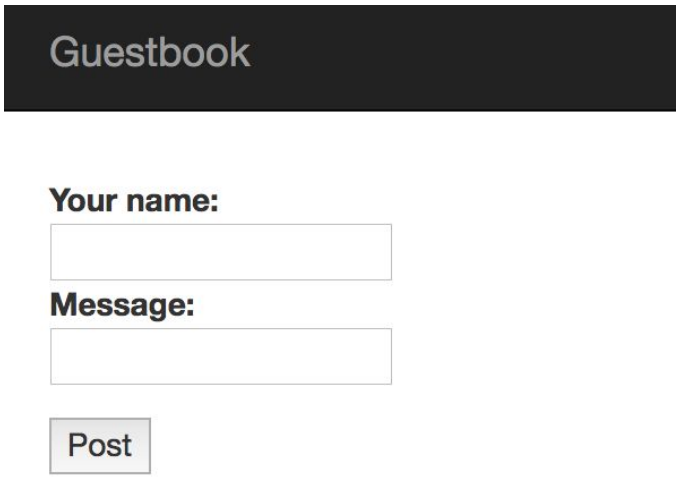
1. Test it out.

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

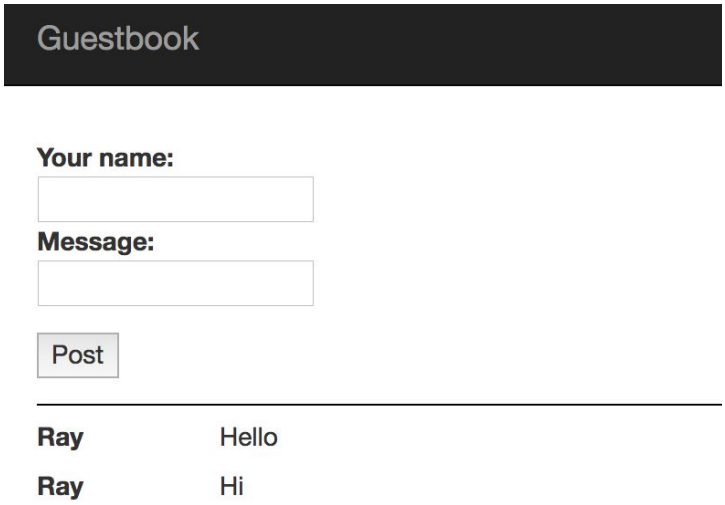
2. This should launch the frontend application on port 8080. Use Cloud Shell's web preview for port 8080.



This will open a new browser tab.



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



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.

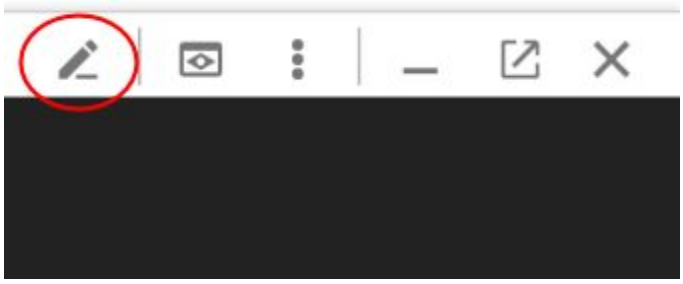
```
$ curl -s http://localhost:8081/guestbookMessages
```

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

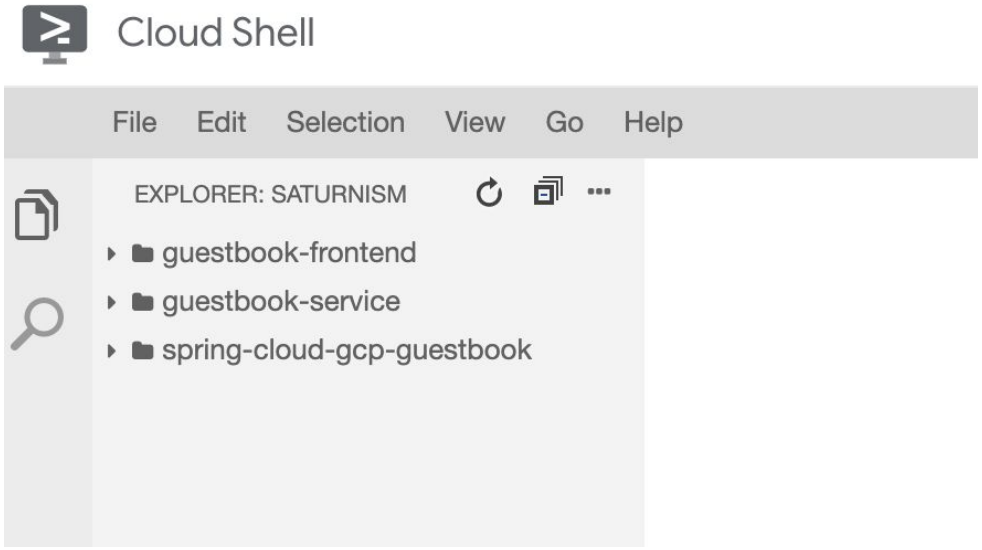
```
$ curl -s http://localhost:8081/guestbookMessages \
  | jq -r '._embedded.guestbookMessages[] | {name: .name, message: .message}'
```

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 web-based 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 2a - 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 use the Spring Cloud GCP Cloud SQL Starter. The starter dependency is included in Spring Cloud BOM.

`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-sql-mysql</artifactId>
            </dependency>
      </dependencies>

      ...
</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`

```
server.port=${PORT:8081}

# Disable Cloud SQL for development
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 Backend

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 test
```

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

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

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

```
… 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
```

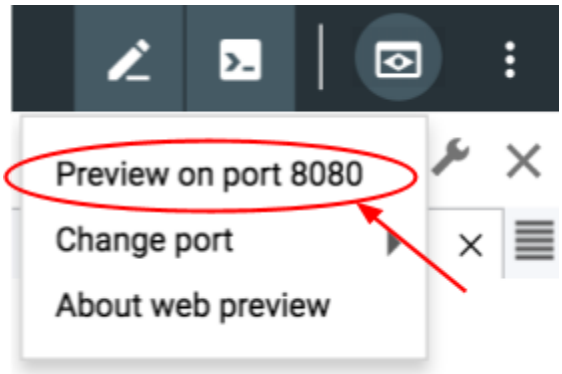
6. You can also list all the messages.

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

8. 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;
```

9. Finally, if your frontend is running, you can use the Web Preview feature to see the frontend, and interact with the backend.



Lab 2b - 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 -p 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="$(<~/guestbook-service/db/spanner.ddl)"
```

Task 3 - Add Spring Cloud GCP Cloud Spanner Starter

1. Update the Guestbook Service's `pom.xml` file to remove the Cloud SQL starter, and add 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>
          ...

          <!-- DELETE Spring Data JPA -->
          <dependency>
               <groupId>org.springframework.boot</groupId>
               <artifactId>spring-boot-starter-data-jpa</artifactId>
          </dependency>

          <!-- DELETE Cloud SQL Starter -->
          <dependency>
               <groupId>org.springframework.cloud</groupId>
               <artifactId>spring-cloud-gcp-starter-sql-mysql</artifactId>
          </dependency>

          <!-- DELETE HSQL -->
          <dependency>
               <groupId>org.hsqldb</groupId>
               <artifactId>hsqldb</artifactId>
               <scope>runtime</scope>
          </dependency>

          <!-- Add Spring Cloud GCP Spanner -->
          <dependency>
               <groupId>org.springframework.cloud</groupId>
               <artifactId>spring-cloud-gcp-starter-data-spanner</artifactId>
          </dependency>
```

```
        ...
    </dependencies>

    ...
</project>
```

Task 4 - Update configuration

1. There is no Cloud Spanner emulator. That means both Dev and Prod will always need a real Cloud Spanner instance running. For the purpose of this lab, we'll use the same Spanner instance. Add the Spanner configuration to `application.properties`.

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

```
server.port=${PORT:8081}

# Add Spanner configuration
spring.cloud.gcp.spanner.instance-id=guestbook
spring.cloud.gcp.spanner.database=messages

# Delete Cloud SQL Configuration
# Disable Cloud SQL for development
spring.cloud.gcp.sql.enabled=false

...
```

2. Remove Database configuration from `application-cloud.properties`, and delete the Cloud SQL configurations.

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

```
# Delete Cloud SQL Configuration
spring.cloud.gcp.sql.enabled=true
spring.cloud.gcp.sql.database-name=messages
spring.cloud.gcp.sql.instance-connection-name=...
spring.datasource.hikari.maximum-pool-size=5

...
```

Task 5 - Update Backend code

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;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

@Data
@Table(name = "guestbook_message")
@JsonIgnoreProperties(value={"id"}, allowSetters = false)
public class GuestbookMessage {
      @PrimaryKey
      @Id
      private String id;

      private String name;

      private String message;

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

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

Task 6 - Run Backend

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.

```
$ ./mvnw spring-boot:run
```

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`.

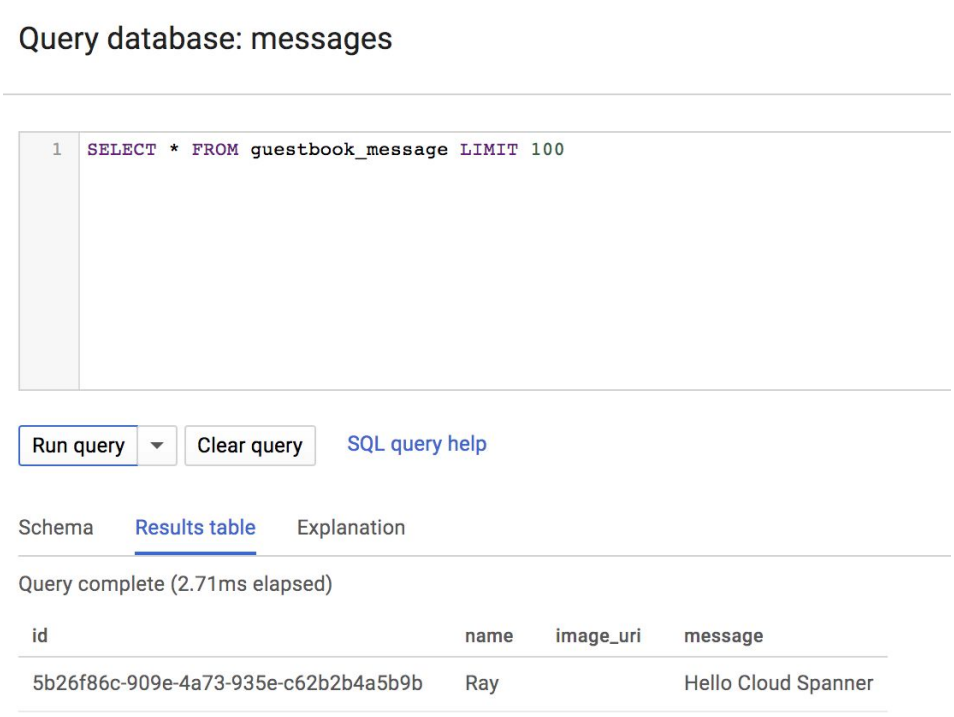
```
$ curl -XPOST -H "content-type: application/json" \
  -d '{"name": "Ray", "message": "Hello Cloud Spanner"}' \
  http://localhost:8081/guestbookMessages
```

5. You can also list all the messages.

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

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

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



Lab 3 - 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 - Configure Stackdriver Trace for the Guestbook Service Backend

1. Add the Spring Cloud GCP Trace starter to `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>
```

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

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

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

3. 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=1.0
spring.sleuth.scheduled.enabled=false
```

Note: By default, Spring Cloud Sleuth auto-configuration instruments executor beans, which causes many traces with the name `async` to appear in Stackdriver Trace. This is especially a problem because our starter comes with an executor. To avoid this noise, please disable automatic instrumentation of executors via `spring.sleuth.scheduled.enabled=false` in your application configuration

Task 3 - Configure Stackdriver Trace for the Guestbook Service Frontend

1. Add the Spring Cloud GCP Trace starter to `pom.xml`.

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>
```

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

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

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

3. For the cloud profile, we'll enable trace sampling for 100% of the requests. Create the application-cloud.properties.

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

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

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
```

Warning: This will 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 public!

Task 5 - Run

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

2. Stop the existing Guestbook Frontend and restart it.

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

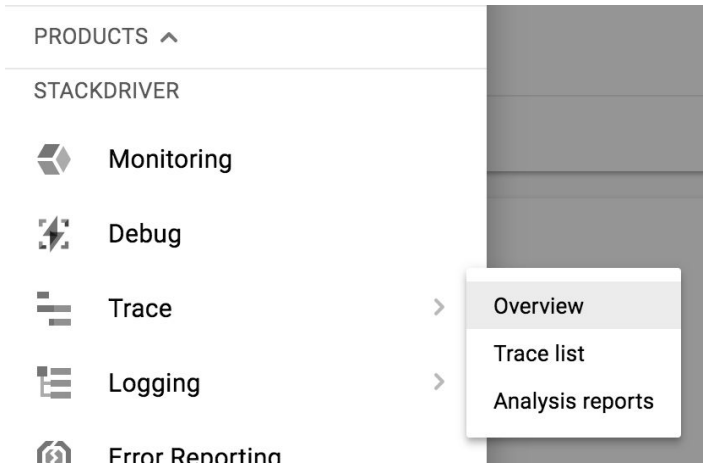
3. Make a request from Cloud Shell web preview. or, make a couple of requests from Cloud Shell.

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

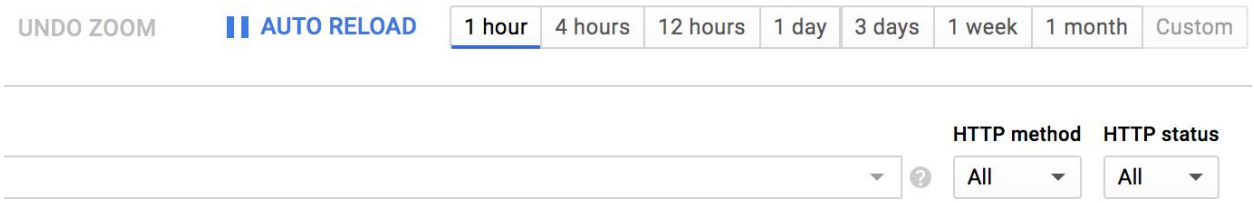
Task 6 - Examine the traces

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.

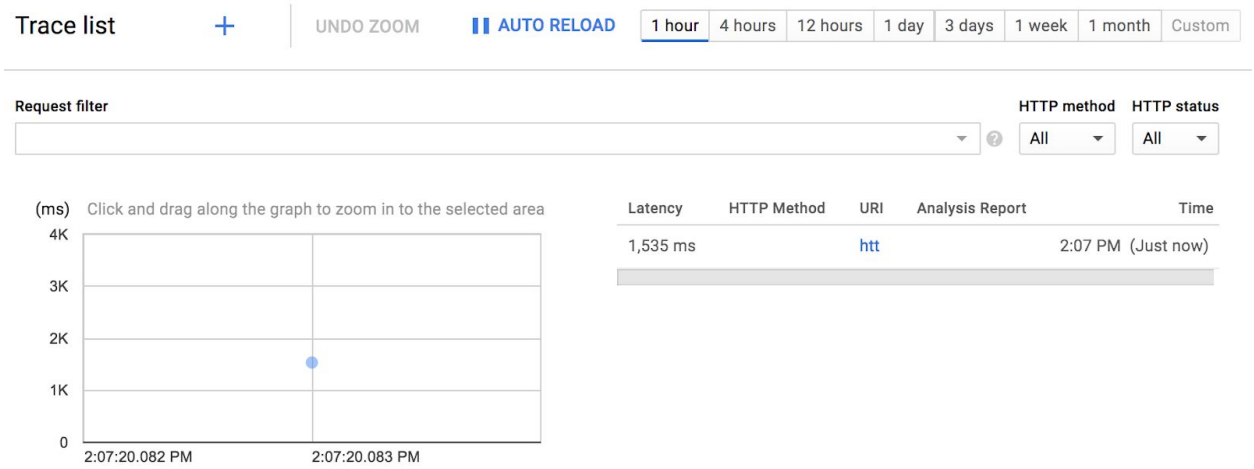
1. In a new browser tab, from the Google Cloud Platform console, navigate to **Stackdriver** → **Trace** → **Trace List**. You should see the traces for the requests you've made.



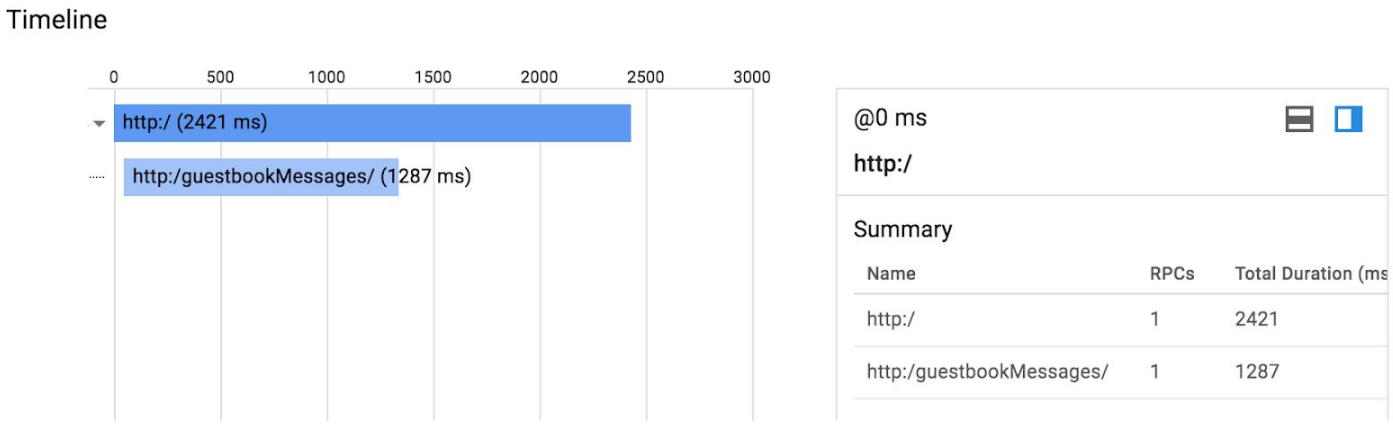
2. On the top, narrow down the time range to 1 hour. By default, **Auto Reload** is on. So as trace data arrives, it will show up in the console.



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



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



Lab 4 - Stackdriver Logging

Duration: 30:00

When running multiple microservices, it's useful to be able to aggregate logs into a centralized logging infrastructure. This is typically done using the ELK (ElasticSearch, LogStash, Kibana) stack. On Google Cloud Platform, you can use Stackdriver Logging.

If you run your application on-premise, you can stream logs to Stackdriver Logging via the API. If you run your application in Google Cloud environments, such as App Engine, Kubernetes Engine, or Cloud Run, you can output the logs to STDOUT and the entries will be automatically aggregated.

Task 1 - Enable Stackdriver Logging API

1. In a new shell tab, enable Stackdriver Logging API first. This may already be enabled by other products you've used.

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

Task 2 - Add Log Messages

1. Add some log messages to the Frontend.

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

```
...
// Use Lombok to inject Slf4J logger
import lombok.extern.slf4j.Slf4j;

@Controller
@SessionAttributes("name")
// Add Slf4j
@Slf4j
public class FrontendController {
```

```
      @PostMapping("/post")
      public String post(@RequestParam String name, @RequestParam String
message, Model model) {
            model.addAttribute("name", name);
            if (message != null && !message.trim().isEmpty()) {
                  // Add a log message at the beginning
                  log.info("Saving message");
                  ...
                  // Add a log message at the end.
                  log.info("Saved message");
            }
            return "redirect:/";
   }

}
```

Task 3 - Configure Stackdriver Logging

For this exercise, you'll only configure Stackdriver Logging for the frontend.

1. Add the Spring Cloud GCP Logging starter to `pom.xml`.

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-logging</artifactId>
            </dependency>
      </dependencies>
      ...
</project>
```

2. To send log entries to Stackdriver Logging, create a `logback-spring.xml` and configure it to use the Stackdriver Logging appender.

guestbook-frontend/src/main/resources/logback-spring.xml

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

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

```
        <root level="INFO">
            <appender-ref ref="STACKDRIVER"/>
            <appender-ref ref="CONSOLE"/>
        </root>
    </springProfile>
    <springProfile name="default">
        <root level="INFO">
            <appender-ref ref="CONSOLE"/>
        </root>
    </springProfile>
</configuration>
```

This configuration uses only the Console appender in development. It uses both Console appender and the Stackdriver appender when using the `cloud` profile.

Task 4 - Run Frontend

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 Frontend and restart it.

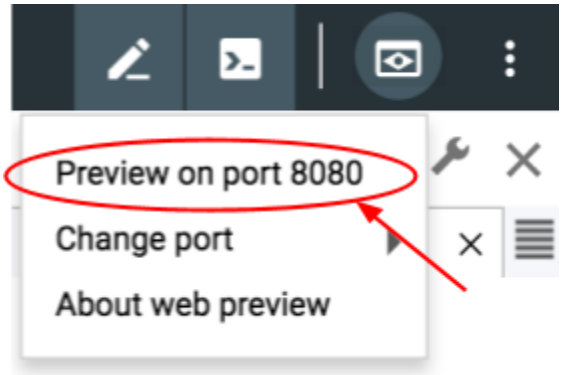
```
$ cd ~/guestbook-frontend
$ ./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-Dspring.profiles.active=cloud \
  -Dspring.cloud.gcp.credentials.location=file:///$HOME/service-account.json"
```

3. Make a request from Cloud Shell web preview. or, make a couple of requests from Cloud Shell.

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

Task 4 - Post a Message

1. Use Web Preview to see the frontend application.



2. Post a message.

3. In the Frontend's output, examine the message:.

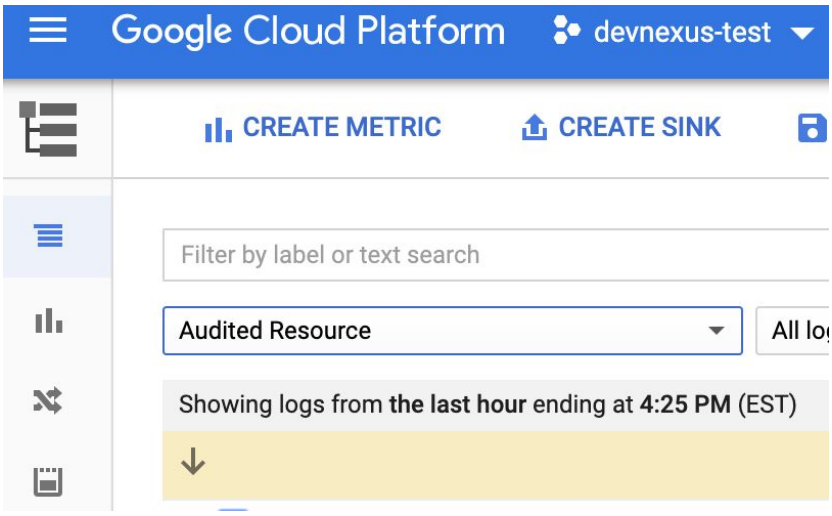
For example:

```
...  INFO [,a2736bfa689daff86061a226984ad0f8,54d617e37b636f48,true] 6103 ---
[nio-8080-exec-2] com.example.frontend.FrontendController  : Saving message
...  INFO [,a2736bfa689daff86061a226984ad0f8,54d617e37b636f48,true] 6103 ---
[nio-8080-exec-2] com.example.frontend.FrontendController  : Saved message
```

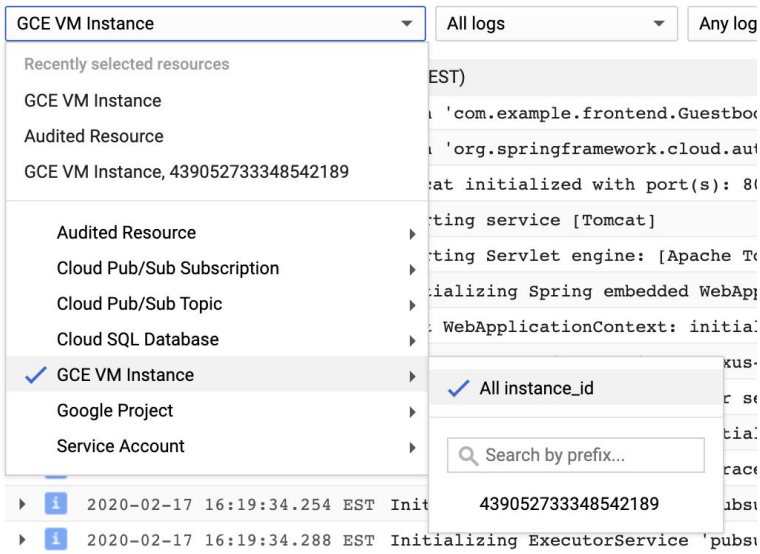
Because you enabled Tracing, each log message is also associated with the trace ID, span ID, and sampled bit (which is always true, because we set the probability to 1.0 - 100%).

Task 5 - Examine the logs

1. In a new browser tab, from the Google Cloud Platform console, navigate to **Stackdriver → Logging → Log Viewer**.



2. In the log category dropdown, select **GCE VM Instance → All instance_id**. This is because the application is running the Cloud Shell, which is a GCE VM instance. If you run it from a different environment, the log message will be in a different location.

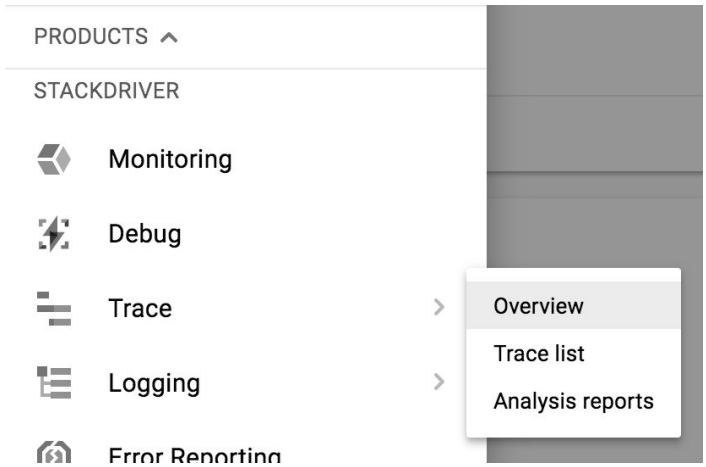


3. You can now see the log message in Stackdriver Logging. Notice that if you expanded the "Saved message", you will see the `trace` and `spanId` attributes.

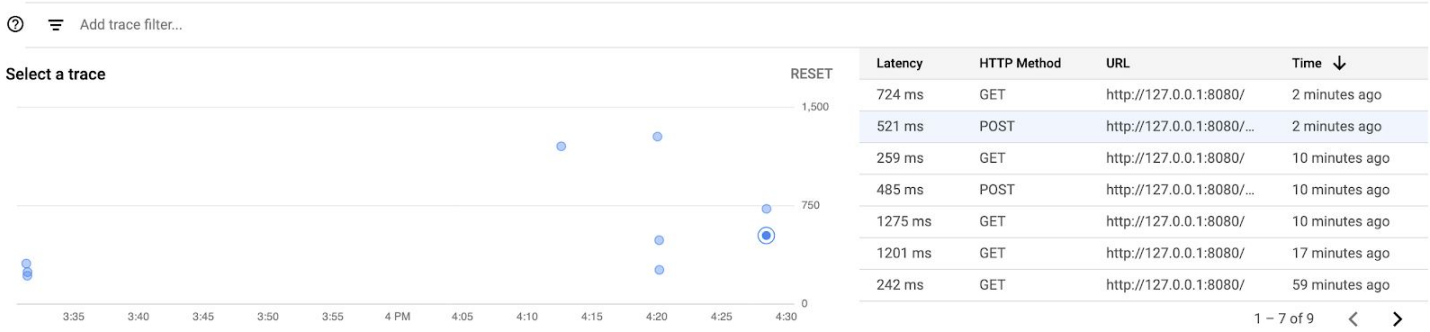
Task 5 - Examine the traces

Each log entry is associated with the corresponding Trace ID and Span ID. Stackdriver Logging can automatically correlate the traces to the log messages too.

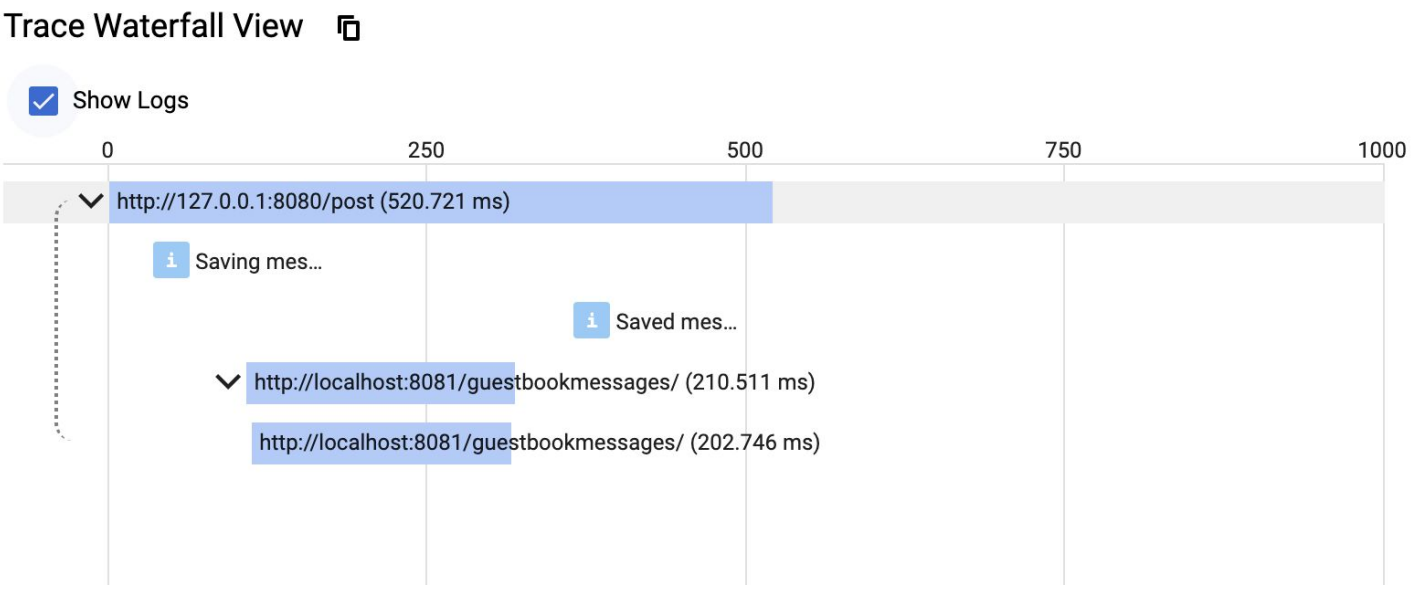
1. In a new browser tab, from the Google Cloud Platform console, navigate to **Stackdriver → Trace → Trace List**.



2. Find the latest trace that posts to the endpoint, and click the **blue** dot to see the trace detail.



3. Click on **Show Logs** to see the log messages correlated to the trace.



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

Task 5 - Run Frontend

1. Restart the Guestbook Frontend.

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

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, clients 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 pubsub subscriptions create messages-subscription-1 \
  --topic=messages
```

2. Pull messages from the subscription.

```
$ gcloud 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 to the creation of a subscription. Go back to the frontend application and post another message, and then pull the message again.

```
$ gcloud 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 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=web,cloud-gcp-pubsub \
  -d baseDir=message-processor | tar -xzvf -
```

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

2. 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>
            ...
      </dependencies>
...
```

3. 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.ApplicationRunner;
import org.springframework.cloud.gcp.pubsub.core.*;

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

      public static void main(String[] args) {
```

```
                SpringApplication.run(DemoApplication.class, args);
        }
}
```

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

`message-processor/src/main/resources/application.properties`

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

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

```
$ cd ~/message-processor
$ ./mvnw 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
                  ...
                  // At the very end, publish the message
                  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;

// Add Spring Integration imports
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 Frontend

1. Restart the Guestbook Frontend.

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

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

If you used the gcloud command to pull the message, you'll also notice that the trace information is automatically added to the message attributes (headers):

```
┌──────┬────────────────┬──────────────────────────────────────────────────┐
│ DATA │   MESSAGE_ID   │                    ATTRIBUTES                    │
├──────┼────────────────┼──────────────────────────────────────────────────┤
│ ...  │ 994932513945890 │ X-B3-ParentSpanId=41f34d29b630a9bb               │
│      │                │ X-B3-Sampled=1                                   │
│      │                │ X-B3-SpanId=51bb189875cd5960                     │
│      │                │ X-B3-TraceId=62f325a50936f7c795a7a0c954c1f2f5    │
│      │                │ spanId=51bb189875cd5960                          │
│      │                │ spanParentSpanId=41f34d29b630a9bb                │
│      │                │ spanSampled=1                                    │
│      │                │ spanTraceId=62f325a50936f7c795a7a0c954c1f2f5     │
└──────┴────────────────┴──────────────────────────────────────────────────┘
```

If you use Spring Integration to receive messages, the trace context will automatically propagate.

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 `FrontendContoller` to accept the file.

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

```
package com.example.frontend;

import ...;
...

// Add imports for Resource
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
                  GuestbookMessage payload = new GuestbookMessage();
                  payload.setName(name);
                  payload.setMessage(message);
                  // Store the generated file name in the database
                  payload.setImageUri(filename);
                  client.add(payload);

                  ...

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

2. 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>
```

Task 4 - Run Frontend

1. Restart the Guestbook Frontend.

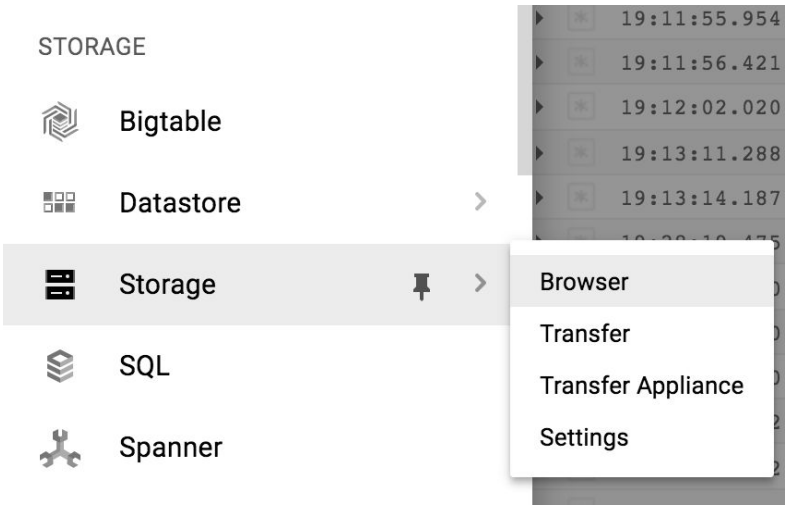
```
$ cd ~/guestbook-frontend
$ ./mvnw spring-boot:run \
  -Dspring-boot.run.jvmArguments="-Dspring.profiles.active=cloud \
  -Dspring.cloud.gcp.credentials.location=file:///$HOME/service-account.json"
```

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

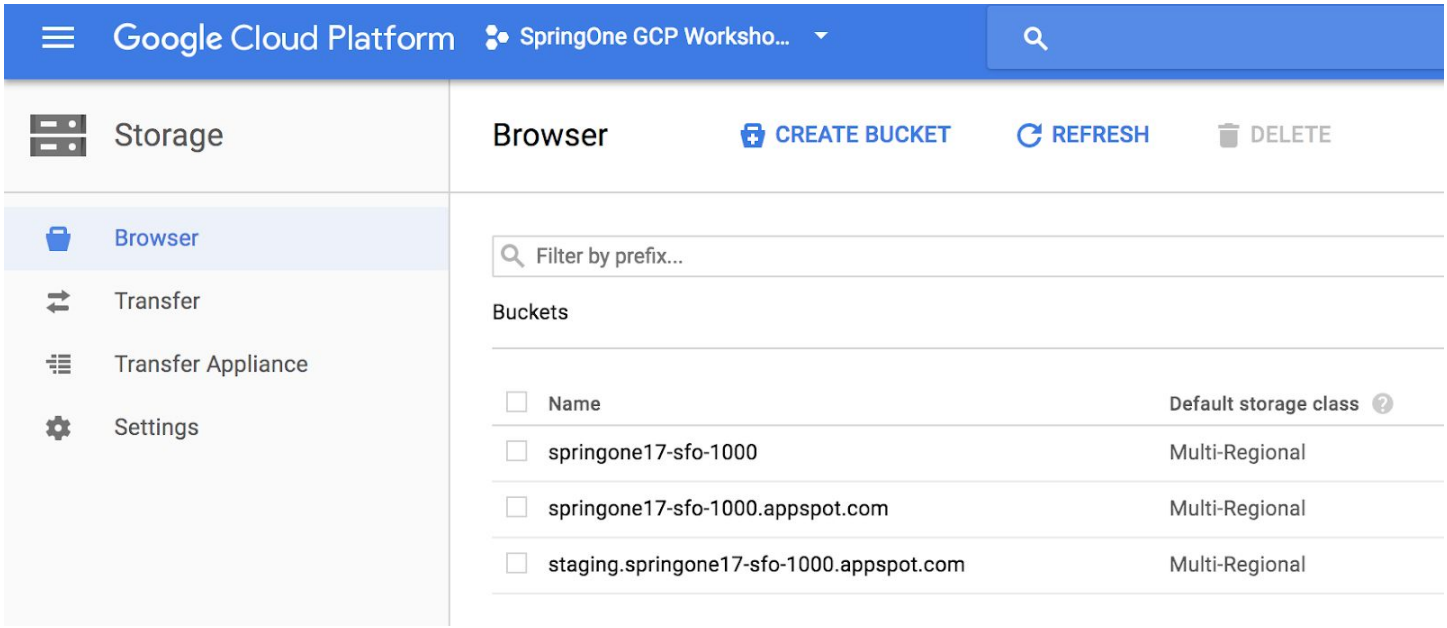
3. Validate that the image was uploaded. From the command line.

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

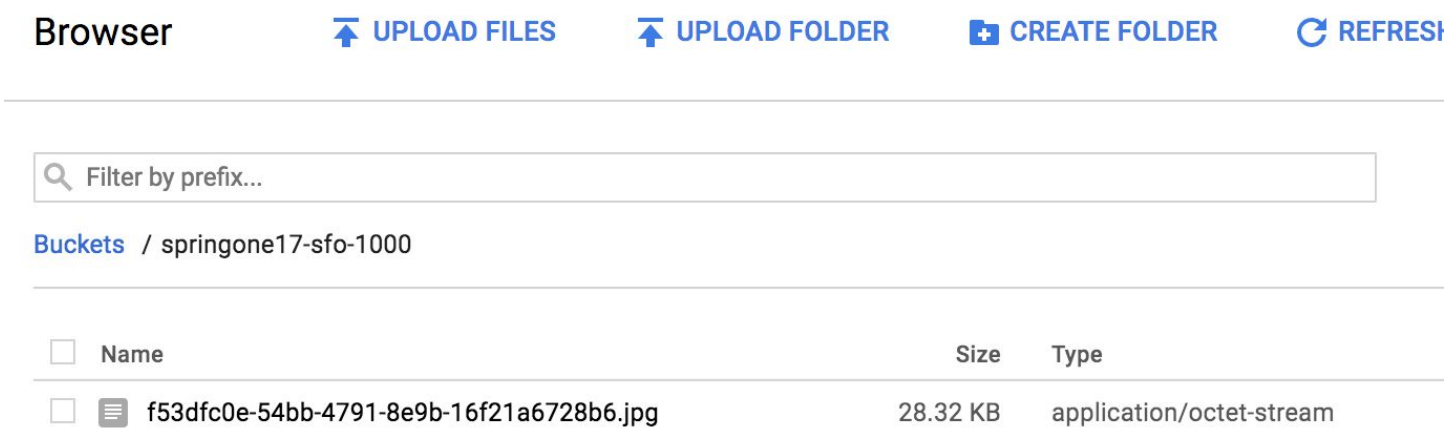
4. From the Google Cloud Platform Console, browse to **Storage** → **Storage** → **Browser**.



6. Navigate to your bucket.



Where you should see the uploaded file.



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>
            <img th:src="'/image/' + ${message.imageUri}"
                alt="image" height="40px"
                th:unless="${#strings.isEmpty(message.imageUri)}"/>
        </div>
    </div>

</div>

...

</body>
</html>
```

Task 6 - Run Frontend

1. Restart the Guestbook Frontend.

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

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

Lab 7 - Vision API

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

Task 2 - 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. Inject `CloudVisionTemplate` into the controller

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

```
package com.example.frontend;

import ...;
...


// Add Vision API imports
import org.springframework.cloud.gcp.vision.CloudVisionTemplate;
import com.google.cloud.vision.v1.Feature.Type;
import com.google.cloud.vision.v1.AnnotateImageResponse;

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

```
      ...

      @Autowired
      private CloudVisionTemplate visionTemplate;

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

      ...
}
```

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 (OutputStream os = resource.getOutputStream()) {
                        ...
                  }

                  // After writing to GCS, analyze the image.
                  AnnotateImageResponse response = visionTemplate
                        .analyzeImage(resource, Type.LABEL_DETECTION);
                  log.info(response.toString());
            }

            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 Frontend

1. Restart the Guestbook Frontend.

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

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 the 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 deploy the application into the same region.

Task 2 - Deploy Guestbook Frontend

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>2.2.0</version>
                      <configuration>
                            <version>1</version>
                            <deploy.projectId>GCLOUD_CONFIG</deploy.projectId>
                      </configuration>
                </plugin>
           </plugins>
     </build>
...
```

2. Create a appengine directory in Guestbook Frontend.

```
$ mkdir -p ~/guestbook-frontend/src/main/appengine
```

3. Add the `app.yaml` that is needed to deploy to App Engine.

`guestbook-frontend/src/main/appengine/app.yaml`

```
runtime: java11
instance_class: B4_1G
manual_scaling:
  instances: 2
env_variables:
  SPRING_PROFILES_ACTIVE: cloud
  # REPLACE PROJECT_ID with your project ID!
  MESSAGES_ENDPOINT:
https://guestbook-service-dot-PROJECT_ID.appspot.com/guestbookMessages
```

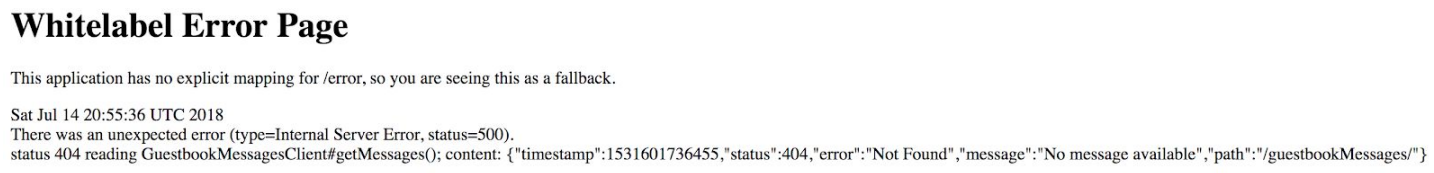
4. Use Maven to deploy the application.

```
$ cd ~/guestbook-frontend
$ ./mvnw package 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
...
```

5. 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!
```

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



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

Task 3 - Deploy Guestbook Service

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>2.2.0</version>
                        <configuration>
                              <version>1</version>
                              <deploy.projectId>GCLOUD_CONFIG</deploy.projectId>
                        </configuration>
                  </plugin>
            </plugins>
      </build>
...
```

2. Create a appengine directory in Guestbook Service.

```
$ mkdir -p ~/guestbook-service/src/main/appengine
```

3. Add the `app.yaml` that is needed to deploy to App Engine.

`guestbook-service/src/main/appengine/app.yaml`

```
runtime: java11
service: guestbook-service
instance_class: B4_1G
```

```
manual_scaling:
  instances: 2
env_variables:
  SPRING_PROFILES_ACTIVE: cloud
```

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.

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

```
$ cd ~/guestbook-service
$ ./mvnw package 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
...
```

5. 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!
```

6. 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"
    }
  }
}
```

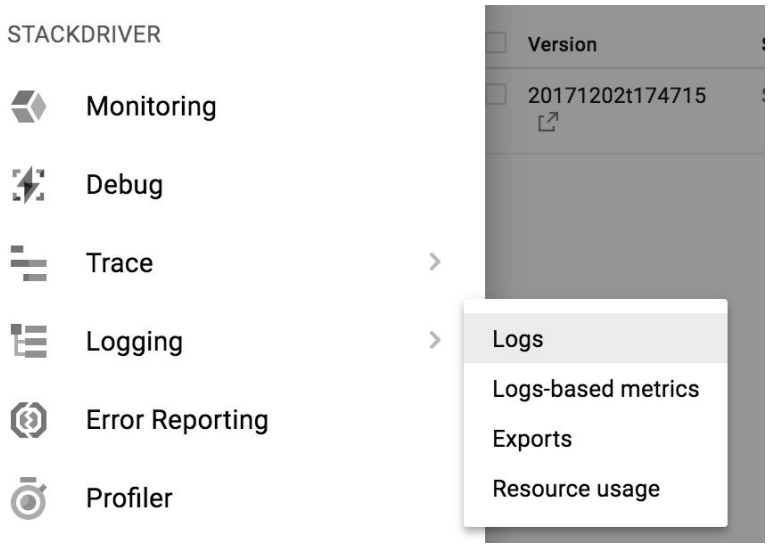
7. Follow the `href` links to see all of the messages, e.g.: https://guestbook-service-dot-PROJECT.appspot.com/guestbookMessages, and see the past messages you created (since it's connecting to the same CloudSQL instance.

```
{
  "_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"
```

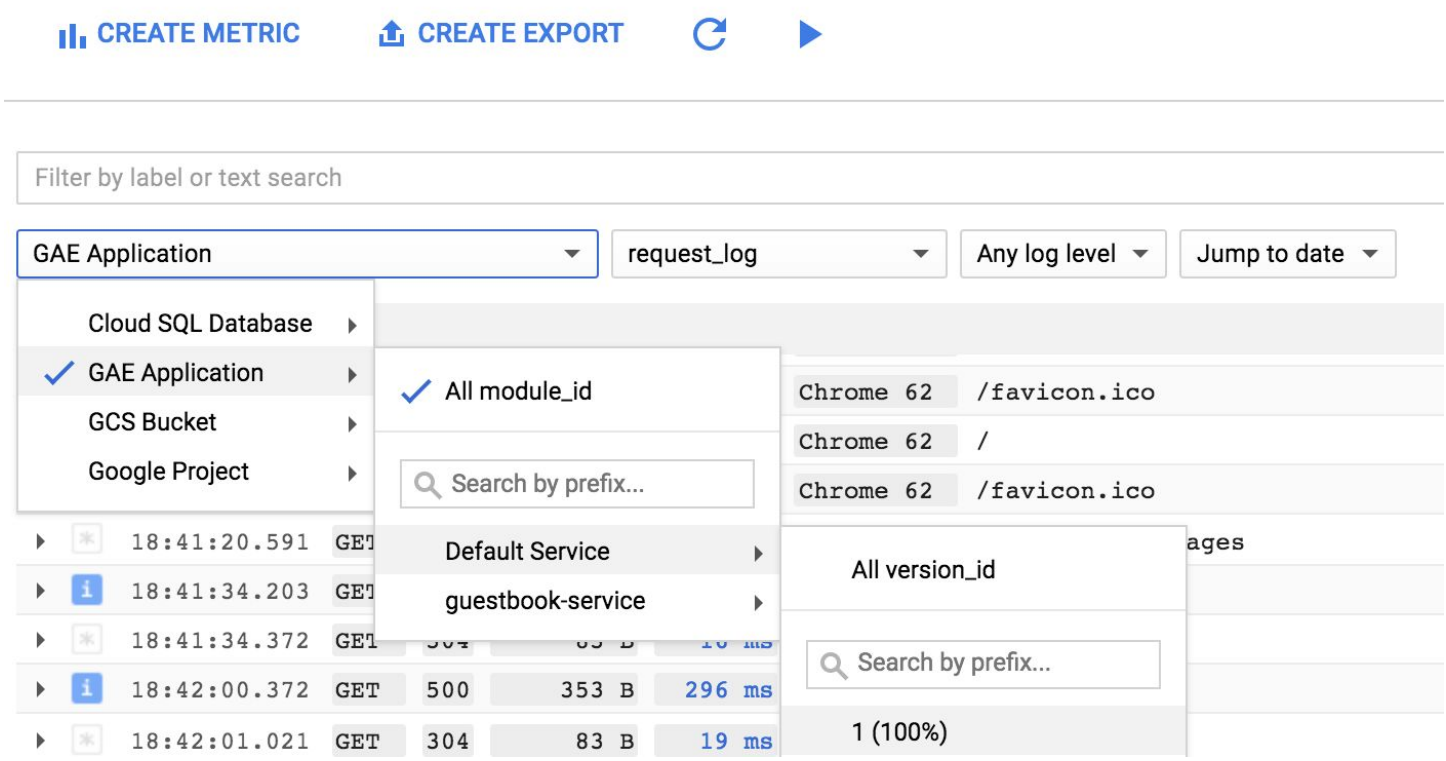

Task 4 - Logging

1. Open a new browser tab and navigate to the Google Cloud Platform console.

2. Navigate to **Stackdriver → Logs**.

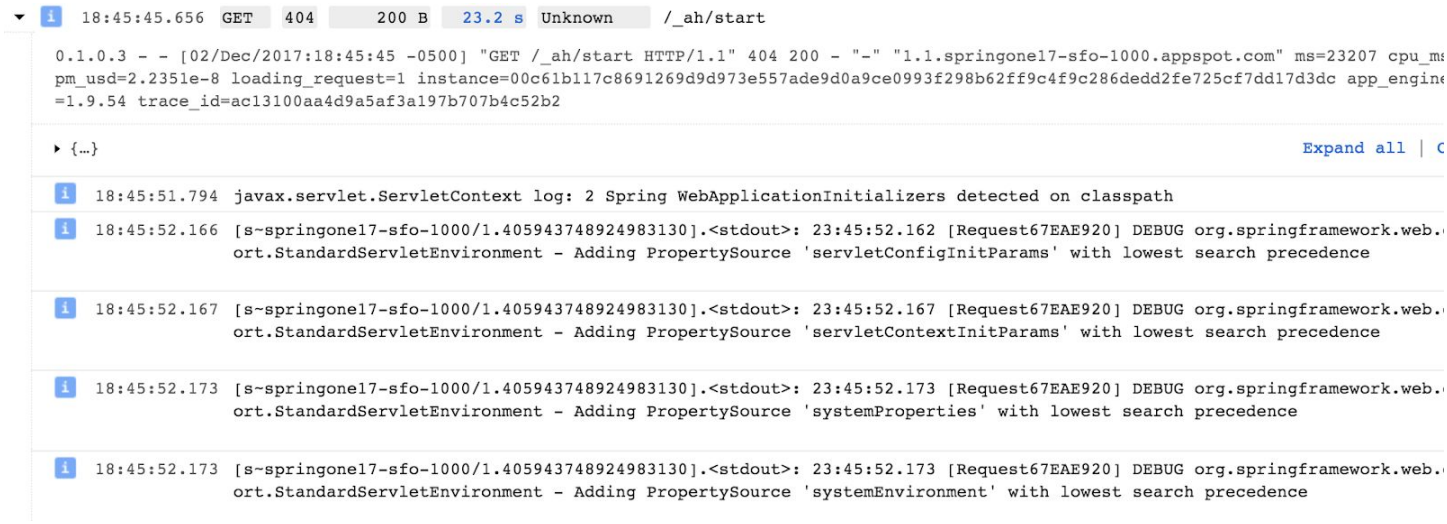


3. In the log drop down, select **GAE Application → Default Service → 1 (100%)**.



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.

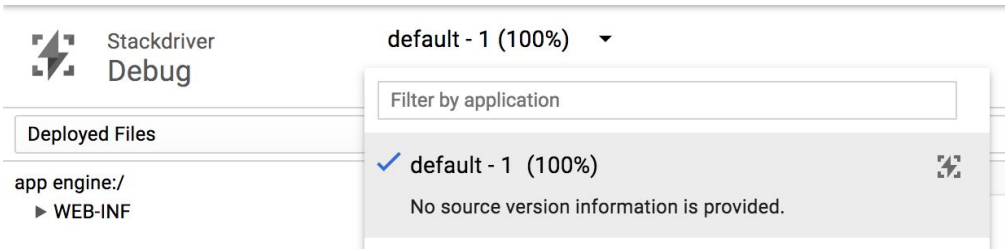
5. Expand one of the entries and see what's there.



Task 5 - Debugging

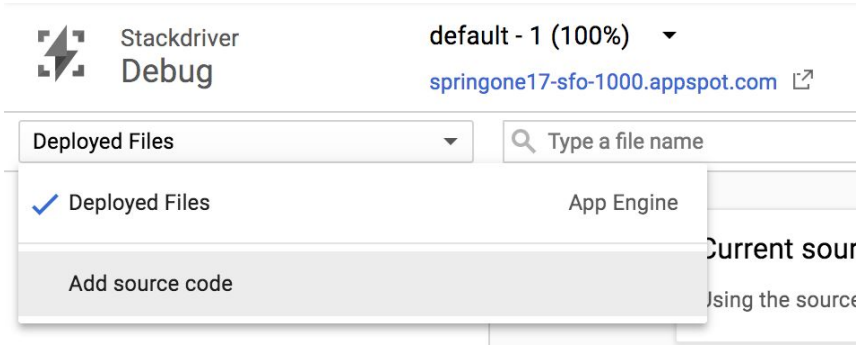
1. Navigate to **Stackdriver** → **Debug**. On the top, you can see the different App Engine deployments currently running.

2. Select the **default - 1 (100%)** from the drop down.



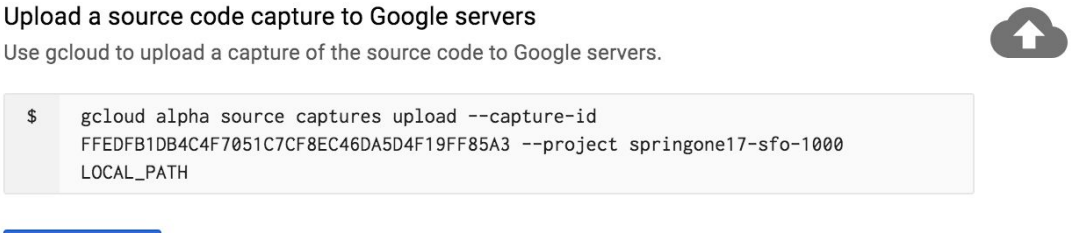
3. However, there is no source code that we can use for debugging.

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.



7. Enable Source Repository API.

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

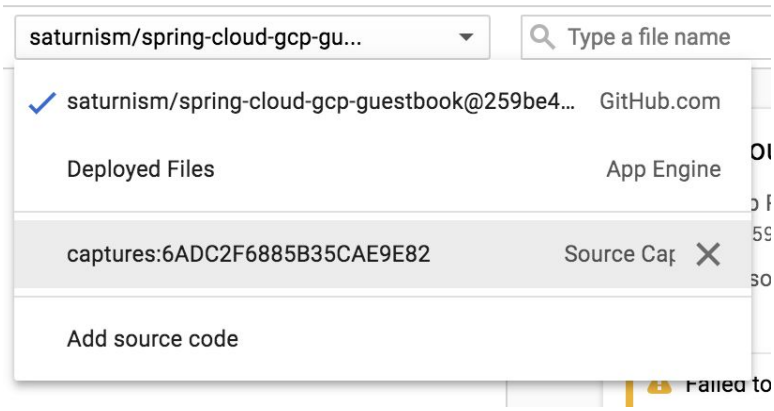
8. Create a Source Repository for Source Capture

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

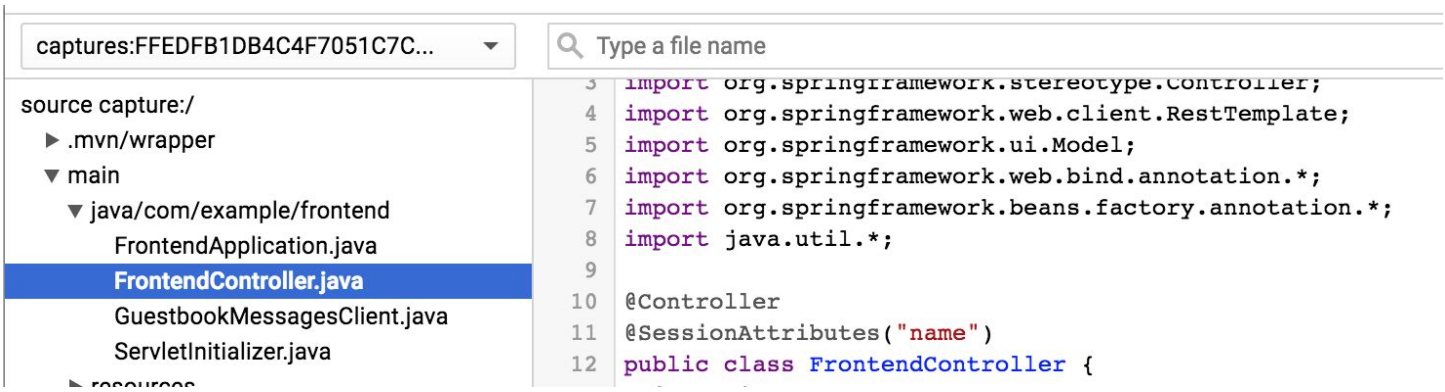
9. Go to Cloud Shell, and use the command line to upload the `guestbook-frontend` source. Please replace the LOCAL_PATH with the actual location of the code.

```
$ cd ~/guestbook-frontend
$ gcloud beta debug source upload --project=... --branch=... ./src/
```

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

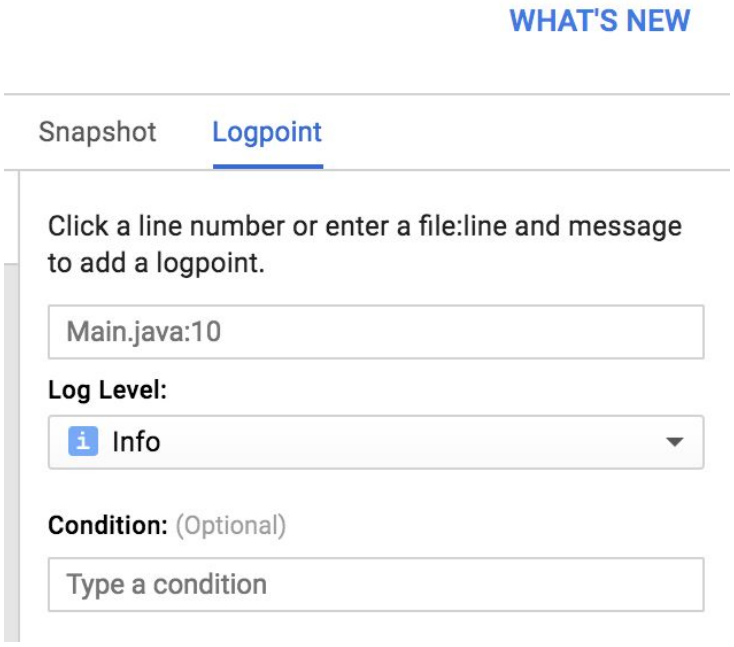


11. Navigate the source and open `FrontendController.java`:

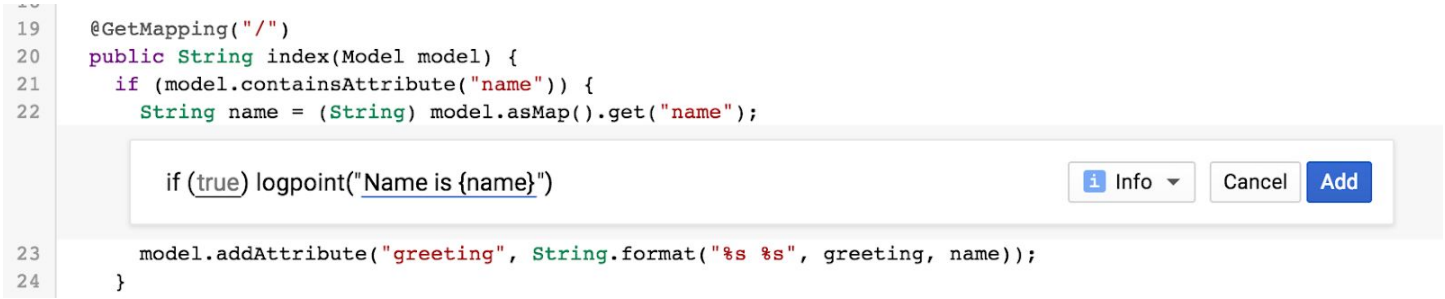


From here, we can do amazing things! For example, add a new log message!

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



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

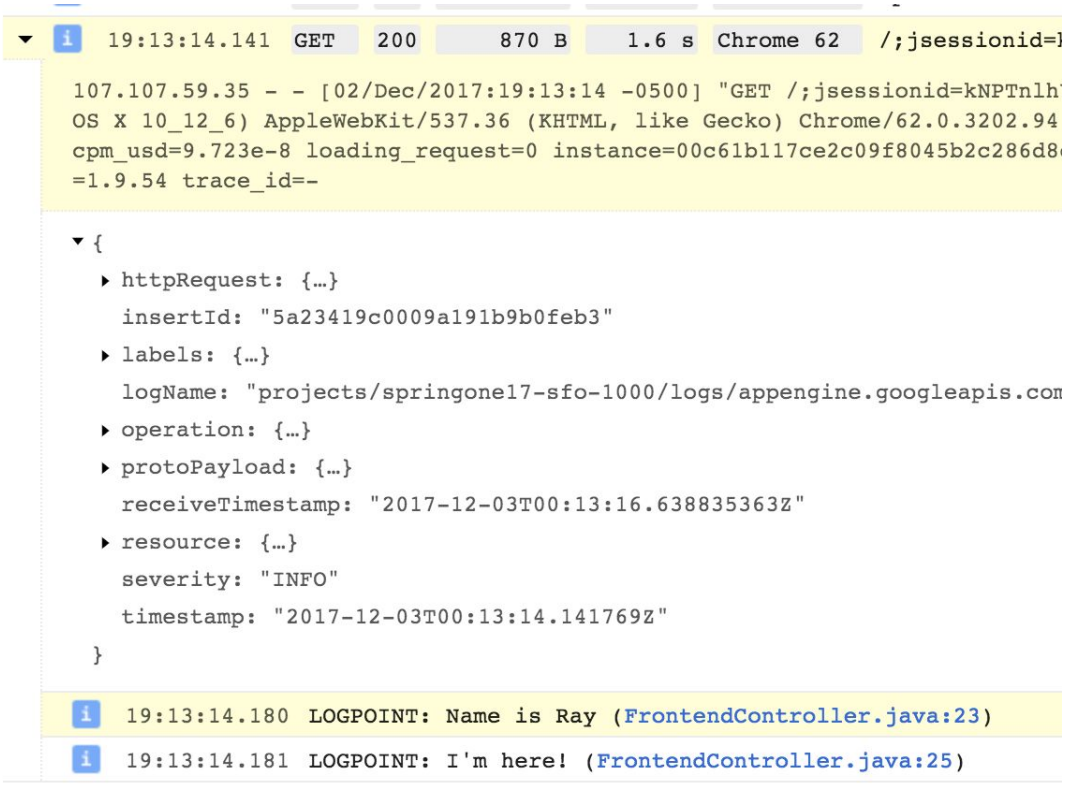


In this example, it'll print the value of the local variable called `name`.

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

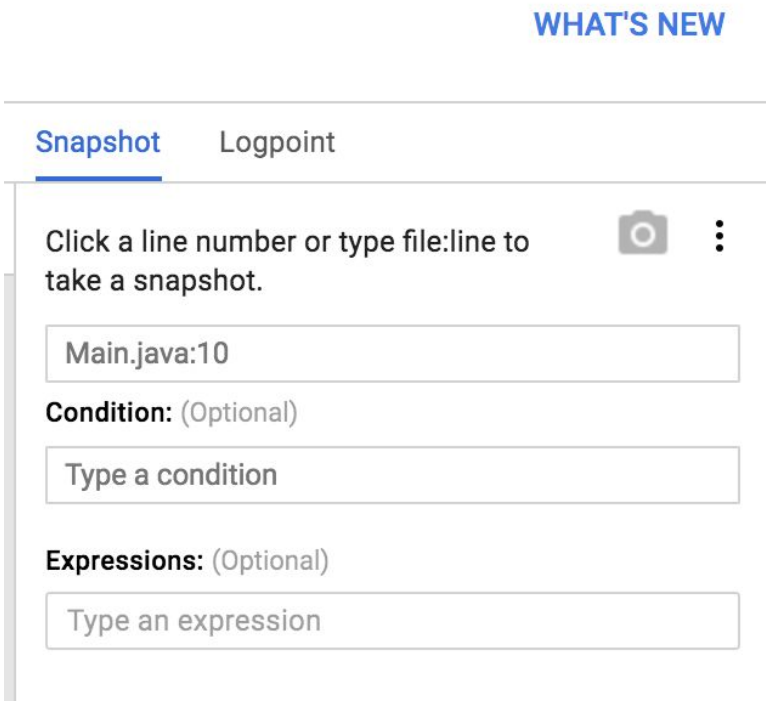
15. Browse to the site and enter the name/message to trigger the code.

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



17. 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.

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



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

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

20. Browse the page again.

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

22. 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 6 - 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 9 - Containerize

Using Jib is the easiest way to containerize your Java application without having to use a Dockerfile! Writing a good Dockerfile is not easy - it takes a lot of discipline and applying best practices. Jib implements many of these best practices.

Task 1 - Enable Container Registry API

1. Enable Container Registry API.

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

Task 2 - Containerize Backend

1. Add Jib plugin to the backend application

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>jib-maven-plugin</artifactId>
                        <version>2.0.0</version>
                        <configuration>
                              <to>
                                    <!-- Replace PROJECT_ID! -->
                                    <image>gcr.io/PROJECT_ID/guestbook-service</image>
                              </to>
                        </configuration>
                  </plugin>

            </plugins>
      </build>
...
```

2. Build the container

```
$ cd ~/guestbook-service
$ ./mvnw compile jib:build
```

Task 3 - Containerize Frontend

1. Add Jib plugin to the frontend application

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>jib-maven-plugin</artifactId>
                        <version>2.0.0</version>
                        <configuration>
                              <to>
```

```
                                    <!-- Replace PROJECT_ID! -->
                                    <image>gcr.io/PROJECT_ID/guestbook-frontend</image>
                                </to>
                            </configuration>
                        </plugin>

                </plugins>
            </build>
...
```

2. Build the container

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

Lab 10 - Cloud Run

Task 1 - Enable Cloud Run API

1. Enable Cloud Run API

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

Task 2 - Deploy

1. Deploy backend

```
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')
$ gcloud run deploy guestbook-service \
  --image=gcr.io/$PROJECT_ID/guestbook-service \
  --region=us-central1 \
  --cpu=2 --memory=2Gi \
  --platform=managed \
  --set-env-vars=SPRING_PROFILES_ACTIVE=cloud \
  --allow-unauthenticated
…
Service [guestbook-service] revision [guestbook-service-00001-tab] has been
deployed and is serving 100 percent of traffic at
https://guestbook-service-4yuo2gqdma-uc.a.run.app
# REMEMBER your domain name!
```

2. Deploy Frontend (replacing **YOUR_GUESTBOOK_HOST** with the hostname from the previous command)

```
$ export MESSAGES_HOST=YOUR_GUESTBOOK_HOST
$ gcloud run deploy guestbook-frontend \
  --image=gcr.io/$PROJECT_ID/guestbook-frontend \
```

```
  --region=us-central1 \
  --cpu=2 --memory=2Gi \
  --platform=managed \
  --allow-unauthenticated \
  --set-env-vars=SPRING_PROFILES_ACTIVE=cloud \
  --set-env-vars=MESSAGES_ENDPOINT=https://$MESSAGES_HOST/guestbookMessages
# Remember to replace domain with the value you got previously
…
Service [guestbook-service] revision [guestbook-service-00001-tab] has been
deployed and is serving 100 percent of traffic at ... ← Visit this URL
```

Lab 11 - Kubernetes

Task 1 - 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 container clusters create guestbook-cluster \
    --cluster-version=1.13 \
    --region=us-central1 \
    --num-nodes=4 \
    --machine-type=n1-standard-2 \
    --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:"13", GitVersion:"v1.13.12",
GitCommit:"a8b52209ee172232b6db7a6e0ce2adc77458829f", GitTreeState
:"clean", BuildDate:"2019-10-15T12:12:15Z", GoVersion:"go1.11.13",
Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"13+",
GitVersion:"v1.13.12-gke.25",
GitCommit:"654de8cac69f1fc5db6f2de0b88d6d027bc15828", GitT
reeState:"clean", BuildDate:"2020-01-14T06:01:20Z", GoVersion:"go1.12.11b4",
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
```

Note: In production systems on GKE, you should look into Workload Identity to securely provision and provide credentials to the GKE cluster.

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. Check to see that all pods are up and running

```
$ watch kubectl get pods
...
NAME                                  READY   STATUS    RESTARTS   AGE
guestbook-frontend-6c7df68bd9-8fd9r   1/1     Running   0          51s
guestbook-frontend-6c7df68bd9-knrcr   1/1     Running   0          51s
guestbook-service-685f8f9fdb-d2cbb    1/1     Running   0          51s
guestbook-service-685f8f9fdb-hcspj    1/1     Running   0          51s
```

6. 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`.

Task 5 - Kustomize

Kustomize allows you to quickly re-use and edit existing Kubernetes manifests for different target environments. For example, you may want to deploy 1 replica in Dev environment, 2 replicas in staging, and 10 replicas in Production. Rather than setting those as variables and using a homegrown templating engine, you can use Kustomize to edit these attributes.

1. Download and install Kustomize

```
$ mkdir -p ~/bin
$ cd ~/bin
$ curl -s
"https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install
_kustomize.sh" | bash
$ export PATH=$PATH:$HOME/bin
```

2. Create kustomize directory.

```
$ mkdir -p ~/kustomize
```

3. Create a base directory that has the base manifest,

```
$ mkdir -p ~/kustomize/base
```

4. Copy the existing manifest into the base directory.

```
$ cp ~/kubernetes/* ~/kustomize/base
```

5. Create a kustomization.yaml

```
$ cd ~/kustomize/base
$ kustomize create
$ kustomize edit add resource guestbook-service-deployment.yaml
$ kustomize edit add resource guestbook-frontend-deployment.yaml
```

This will create a kustomization.yaml with the 2 YAMLs as the base manifests.

6. Use Kustomize to combine the 2 manifests into a single deployable manifest.

```
$ kustomize build
```

7. You can also apply it directly to Kubernetes

```
$ kustomize build | kubectl apply -f -
```

8. Copy the service account credentials and use Kustomize to automatically generate the secret.

```
$ cp ~/service-account.json ~/kustomize/base
$ kustomize edit add secret guestbook-service-account \
  --from-file=service-account.json
```

9. When you use Kustomize to build the manifest again, observe that Secret resource is automatically created with the content of the Service Account. You can use Kustomize to generate ConfigMap as well.

```
$ kustomize build

data:
  service-account.json: ...
kind: Secret
metadata:
  name: guestbook-service-account-mk5f6fcfkc
type: Opaque
```

Notice that it also automatically generates a suffix for the secret.

10. Create a staging directory and re-use the base manifest.

```
$ mkdir -p ~/kustomize/staging
$ cd ~/kustomize/staging
$ kustomize create
$ kustomize edit add base ../base
```

11. Create a patch file for the backend service to increase the number of replicas.

kustomize/staging/guestbook-service-scale.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: guestbook-service
```

```
spec:
  replicas: 4
```

12. Add the patch to Kustomization.

```
$ kustomize edit add patch guestbook-service-scale.yaml
```

13. Let's use the default namespace as the development environment, but staging namespace for the staging environment..

```
$ kubectl create ns staging
$ kustomize edit set namespace staging
```

14. When you use Kustomize to build the manifest for staging environment, observe the number of replicas is now 4, and also the namespace is set to staging.

```
$ kustomize build
```

15. Apply the changes to the staging environment.

```
$ kustomize build | kubectl apply -f -
$ kubectl -n staging get pods
```

Lab 12 - Kubernetes Monitoring

Task 1 - Stackdriver Kubernetes Monitoring

In the 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. This may take a few minutes.

3. Click **Dashboard** → **Kubernetes Engine** to view Kubernetes Monitoring dashboard.

Task 2 - Export Prometheus Metrics

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.

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 9000.

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

```
...

management.server.port=9000
management.endpoints.web.exposure.include=*
```

3. To send log entries to Stackdriver Logging, via STDOUT and structured JSON logging, change `logback-spring.xml` to use the `CONSOLE_JSON` appender..

guestbook-frontend/src/main/resources/logback-spring.xml

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

    <springProfile name="cloud">
        <include
resource="org/springframework/cloud/gcp/logging/logback-json-appender.xml"/>
        <root level="INFO">
            <appender-ref ref="CONSOLE_JSON"/>
        </root>
    </springProfile>
```

```
    <springProfile name="default">
        <root level="INFO">
            <appender-ref ref="CONSOLE"/>
        </root>
    </springProfile>
</configuration>
```

3. Rebuild the container.

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

4. Update the Kubernetes manifest to declare the metrics ports.

kustomize/base/guestbook-frontend-deployment.yaml

```
apiVersion: v1
kind: Service
...
---
apiVersion: apps/v1
kind: Deployment
...
spec:
  ...
  template:
    ...
    spec:
      ...
      containers:
      - name: guestbook-frontend
        ...
        ports:
        - name: http
          containerPort: 8080
        - name: metrics
          containerPort: 9000
```

5. Redeploy the manifest.

```
$ cd ~/kustomize/base
$ kustomize edit set namespace default
$ kustomize build | kubectl apply -f -
```

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

```
$ 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-... 9000:9000
```

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

```
$ curl http://localhost:9000/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.

Task 3 - Install Prometheus and Stackdriver Sidecar

Stackdriver Kubernetes Monitoring can monitor Prometheus metrics from the Kubernetes cluster. Install Prometheus support to the cluster.

1. Install a quickstart Prometheus operator.

```
$ kubectl apply -f \
  https://raw.githubusercontent.com/coreos/prometheus-operator/master/bundle.yaml
```

Note: For production installation, you should consider using the Prometheus Operator's Helm Chart. E.g., with Helm v3:

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com
$ kubectl create ns monitoring
$ helm install prometheus-operator stable/prometheus-operator \
  --namespace=monitoring
```

2. Provision Prometheus using the Prometheus Operator.

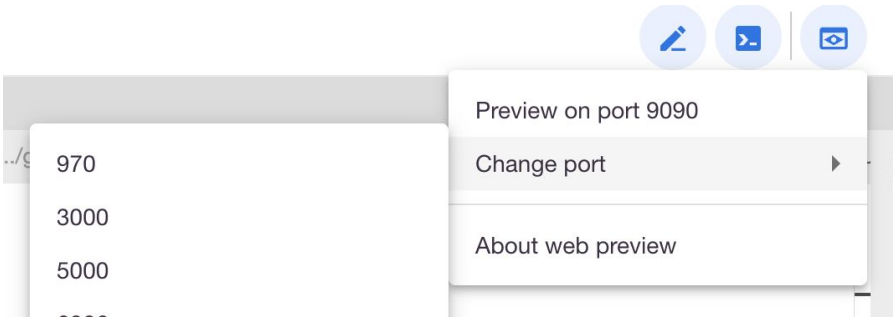
```
$ cd ~/spring-cloud-gcp-guestbook/12-kubernetes-monitoring/prometheus
$ export PROJECT_ID=$(gcloud config list --format 'value(core.project)')

# Make sure the project ID is set
$ echo $PROJECT_ID
$ cat prometheus.yaml | envsubst | kubectl apply -f -
$ kubectl apply -f pod-monitors.yaml
```

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

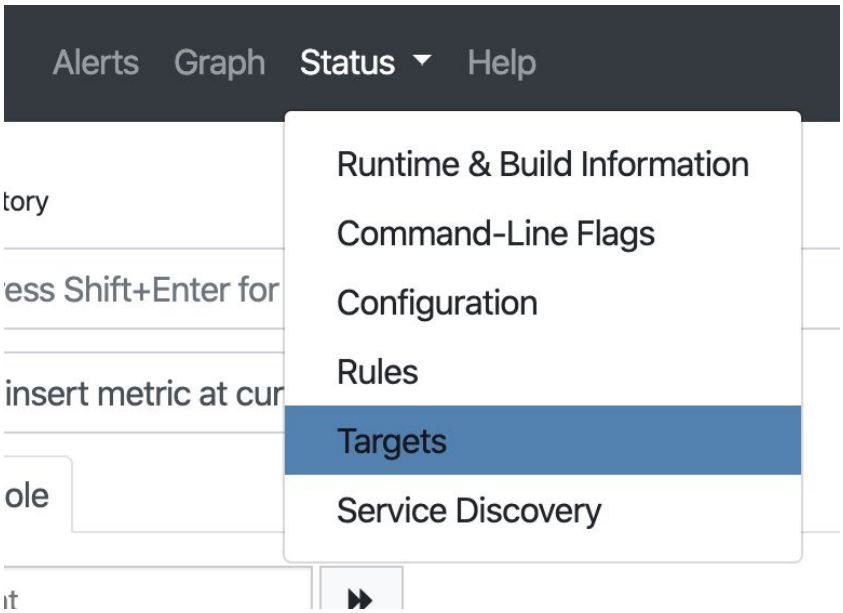
3. Validate Prometheus is running properly and scraping the data. Establish a port forward to Prometheus' port.

```
$ pkill java
$ kubectl port-forward svc/prometheus 9090:9090
```

4. Use Cloud Shell's Web Preview and change port to 9090.



5. In the Prometheus console, select **Status** → **Targets**.



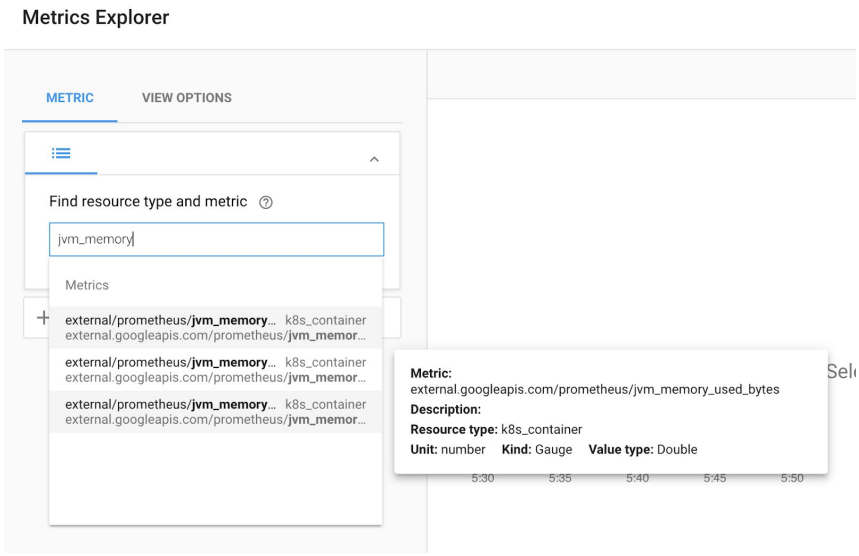
6. Observe that there are 2 targets (2 pods) being scraped for metrics.

Task 4 - Explore the metrics

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

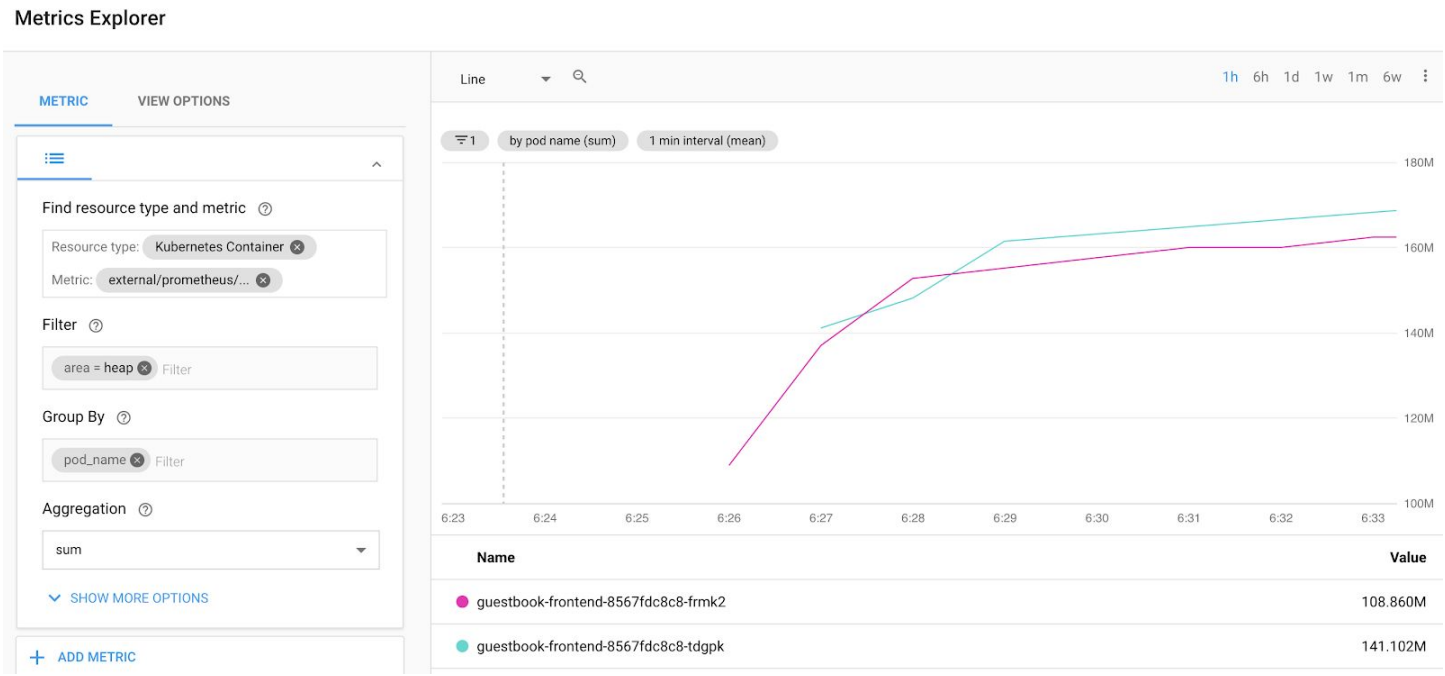
2. Click **Resources** → **Metrics Explorer**.

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



4. Select `jvm_memory_used` to plot the metrics.

5. 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**.



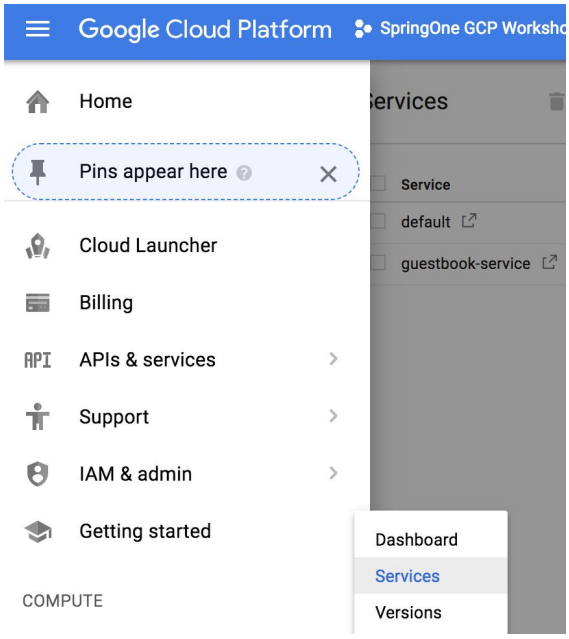
This should build a graph of current Heap usage of the frontend application.

Lab 13 - Clean Up

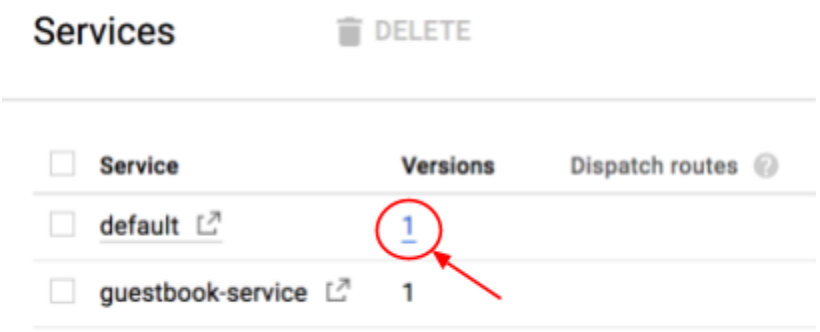
Before you leave, please make sure to clean up the project and remove public facing deployments such as the Guestbook Frontend and service.

Task 1 - Stop App Engine Apps

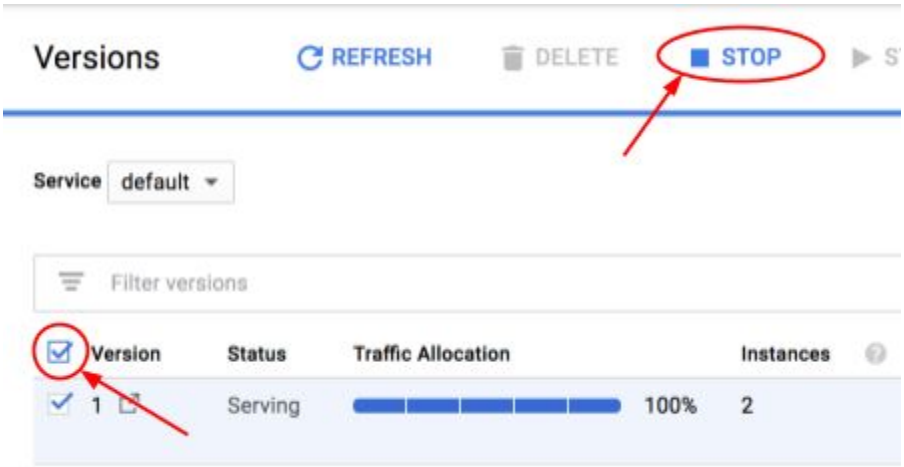
1. Navigate to **Compute → App Engine → Services**



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



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



Task 2 - Delete all Resources

1. Delete all Cloud SQL instances.

2. Delete all Spanner instances.

3. Delete all Kubernetes Engine clusters.