Start Lab 01:30:00

JAVAMS03 Working with Runtime Configurations

1 hour 30 minutes Free ★★★★☆ Rate Lab

Overview

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

In this lab, you modify your application to support a dynamic runtime configuration service.

You can configure your microservices application in several ways. In a production environment, you need a robust mechanism to store and update configuration values that might change dynamically. Typically, you end up with additional configuration servers that you have to make highly available and manage yourself. Cloud Runtime Configuration API enables you to define and store data as a hierarchy of key-value pairs in Google Cloud Platform (GCP). You can use these key-value pairs to dynamically configure services, communicate service states, send notification of changes to data, and share information between multiple tiers of service.

Spring Cloud GCP has a configuration starter that interoperates well with the Spring Cloud Config facility. This starter enables you to integrate runtime configuration capabilities into your applications without having to manually build and manage your own configuration server.

The greeting message produced by the frontend UI in the demo application is configurable through the `greeting` property. You can externalize this functionality into the GCP runtime configuration.

In the frontend guestbook controller source code at `guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`, the `${greeting}` variable is read. The value of that variable defaults to "Hello." You use the Cloud Runtime Configuration API service to override this value in the lab.

Objectives

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

- Use Spring Cloud GCP to add support for Cloud Runtime Configuration API
- Create deployment profiles to selectively enable support for runtime configuration services

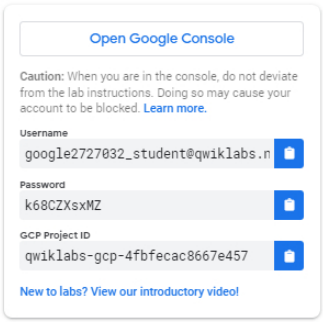
- Enable the dynamic refresh of runtime configuration values in an application
- Use Cloud Runtime Configuration API to create runtime configuration profiles and values
- Use Cloud Runtime Configuration API to dynamically update an application setting

Task 0. Lab Preparation

Access Qwiklabs

How to start your lab and sign in to the Console

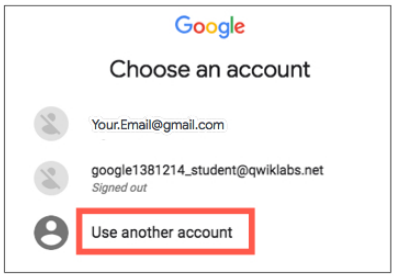
1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is a panel populated with the temporary credentials that you must use for this lab.



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

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

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



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

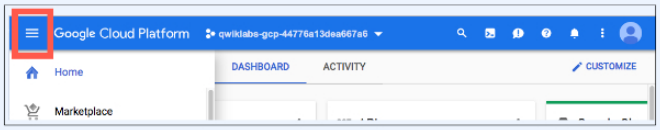
   Important: You must use the credentials from the Connection Details panel. Do not use your Qwiklabs credentials. If you have your own GCP account, do not use it for this lab (avoids incurring charges).

5. Click through the subsequent pages:

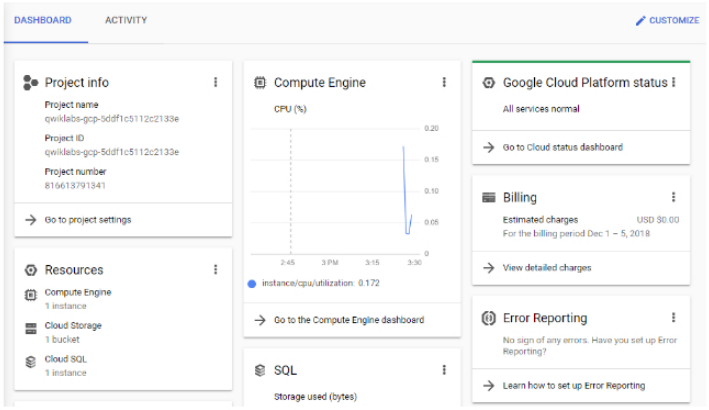
   - Accept the terms and conditions.
   - Do not add recovery options or two-factor authentication (because this is a temporary account).
   - Do not sign up for free trials.

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

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



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



Fetch the application source files

To begin the lab, click the **Activate Cloud Shell** button at the top of the Google Cloud Console. To activate the code editor, click the `Open Editor` button on the toolbar of the Cloud Shell window. You can switch between cloud shell and code editor by using `Open Editor` and `Open Terminal` icon as required.

Note: A Cloud Storage bucket that is named using the project ID for this lab is automatically created for you by the lab setup. The source code for your applications is copied into this bucket when the Cloud SQL server is ready. You might have to wait a few minutes for this action to complete.

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

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

2. Verify that the demo application files were created.

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

3. Copy the application folders to Cloud Shell.

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

4. Make the Maven wrapper scripts executable. Now you're ready to go!

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

Task 1. Add the Spring Cloud GCP Config starter

In this task, you update the frontend application's `pom.xml` file to import the Spring Cloud GCP BOM. You add the milestone repository (the lab version is in this Maven repository). And you include the new starter in the dependency section.

1. Open `guestbook-frontend/pom.xml` in the Cloud Shell code editor.

2. Add the following dependency definition to the `<dependencies>` section:

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-trace</artifactId>
        </dependency>
        <dependency>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-starter-config</artifactId>
            <version>1.2.0.RC2</version>
        </dependency>
```

3. Insert a new section called `<repositories>` at the bottom of the file, after the `<build>` section and just before the closing `</project>` tag.

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

> **Note**
>
> The version for the spring-cloud-gcp-starter-config and the milestones repo are necessary because Runtime Config is in beta.

Task 2. Configure a local profile

When deploying the demo application locally, you want to use a production configuration that includes properties for local services.

In this task, you set a local profile in the default profile for the frontend application.

1. In the `guestbook-frontend/src/main/resources/` folder, **create** a file called `bootstrap.properties` that disables the Spring Cloud GCP starter.

2. Open `guestbook-frontend/src/main/resources/bootstrap.properties` in the Cloud Shell code editor and add the following property:

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

3. Open `guestbook-frontend/src/main/resources/application.properties` and add the following property:

```
```

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

This property allows access to the Spring Boot Actuator endpoint so that you can dynamically refresh the configuration.

> **Note**
>
> The combination of `${spring.cloud.gcp.config.name}_${spring.cloud.gcp.config.profile}` forms `frontend_local`, which is the name of the runtime configuration that you create in a later task.

Task 3. Configure a cloud profile

When deploying the demo application into the cloud, you want to use a production configuration that includes properties for cloud services.

In this task, you use the Spring configuration profile and create a `cloud` profile for the frontend application.

1. In the `guestbook-frontend/src/main/resources/` folder, **create** a file called `bootstrap-cloud.properties` that defines the Spring Cloud Config bootstrapping configuration.

2. Open `guestbook-frontend/src/main/resources/bootstrap-cloud.properties` in the Cloud Shell code editor and add the following new properties:

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

Task 4. Add RefreshScope to FrontendController

By default, runtime configuration values are read only when an application starts. In this task, you add a Spring Cloud Config `RefreshScope` to the frontend application so that runtime configuration values can be updated dynamically without restarting the application. You do this by adding an `@RefreshScope` annotation to the `FrontendController` source file.

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

2. Insert the following lines below the import directives just above `@Controller`:

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

@RefreshScope
```

Task 5. Create runtime configuration

Task 5. Create a runtime configuration

In this task, you enable Cloud Runtime Configuration API and create a runtime configuration value that is used to dynamically update the application.

1. In the Cloud Shell enable Cloud Runtime Configuration API.

```
gcloud services enable runtimeconfig.googleapis.com
```

2. Create a runtime configuration for the frontend application's `local` profile.

```
gcloud beta runtime-config configs create frontend_local
```

A URL for the new runtime configuration similar to the following is displayed:

```
Created [https://runtimec.../v1beta1/projects/.../frontend_local].
```

3. Set a new configuration value for the greeting message.

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

4. Enter the following command to display all the variables in the runtime configuration:

```
gcloud beta runtime-config configs variables list --config-
name=frontend_local
```

5. Enter the following command to display the value of a specific variable.

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

The command displays the value of `greeting`.

Task 6. Run the updated application locally

In this task, you test the changes that you made to the application by testing the Maven build with the default profile and then using the `cloud` profile to restart the frontend service.

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

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

2. Run the backend service application.

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

The backend service application launches on port 8081. The `cloud` profile specified here is configured with the Cloud SQL database enabled. This takes a minute or two to complete and you should wait until you see that the GuestbookApplication is running.

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

3. Open a new Cloud Shell session tab to run the frontend application by clicking the

plus (+) icon to the right of the title tab for the initial Cloud Shell session.

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

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

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

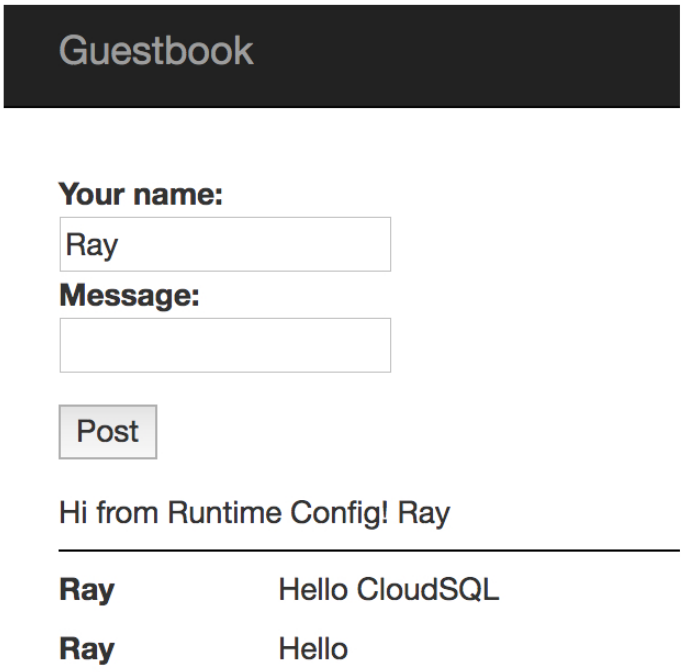
5. Start the frontend application with the `cloud` profile.

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

6. Use the Cloud Shell web preview to browse to the frontend application on port 8080.

7. Enter values for **Your name** and **Message**, and then click **Post** to trigger the Hello response.

8. Verify that the returned greeting message now uses the value you set using the Cloud Runtime Configuration API.



Note

If you use Spring Boot Actuator, you can refresh and reload configurations on the fly. See sample code for how that works. You will test this in the next task.

Task 7. Update and refresh a configuration

In this task, you update a runtime configuration property by using Cloud Runtime Configuration API. And you verify that the application has dynamically refreshed the variable linked to that property.

1. Open a new Cloud Shell tab and update the `greeting` configuration value:

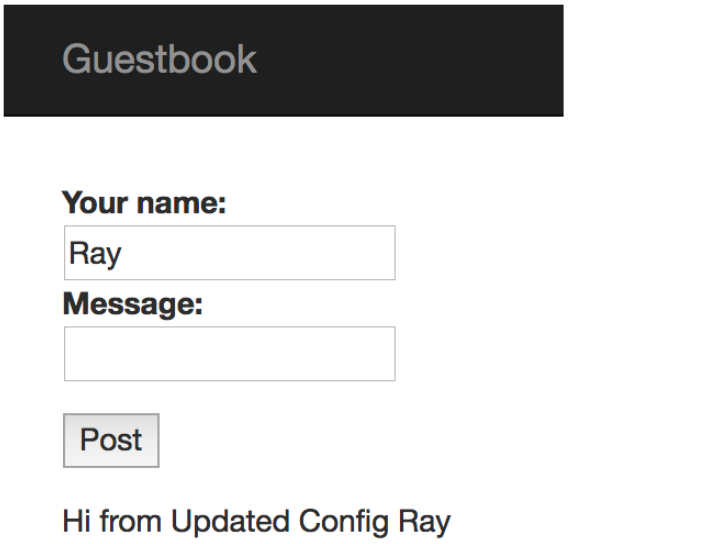
```
gcloud beta runtime-config configs variables set greeting \
  "Hi from Updated Config" \
  --config-name frontend_local
```

2. Use curl to trigger Spring Boot Actuator so that it reloads configuration values from the Runtime Configuration API service.

```
curl -XPOST http://localhost:8080/actuator/refresh
```

3. Post another message to the guestbook application through the frontend application in the Web Preview tab.

   The updated `greeting` response is displayed.



You can also use Spring Boot Actuator to display the current configuration values directly, as well as other information using this syntax.

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

This command prints out the configuration property values in JSON format.

End your lab

When you have completed your lab, click **End Lab**. Qwiklabs removes the resources you've used and cleans the account for you.

You'll be given an opportunity to rate the lab experience. Select the applicable number of stars, type a comment, and then click **Submit**.

The number of stars indicates your rating:

- 1 star = Very dissatisfied
- 2 stars = Dissatisfied
- 3 stars = Neutral
- 4 stars = Satisfied
- 5 stars = Very satisfied

You can close the dialog box if you don't want to provide feedback.

For feedback, suggestions, or corrections, use the **Support** tab.