JAVAMS06 Integrating Cloud Pub/Sub with Spring

2 hoursFree

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 use Spring Integration to create a message gateway interface that abstracts from the underlying messaging system rather than using direct integration with Cloud Pub/Sub.

Using this approach, you can swap messaging middleware that works with on-premises applications for messaging middleware that works with cloud-based applications. This approach also makes it easy to migrate between messaging middlewares.

In this lab, you use Spring Integration to add the message gateway interface and then refactor the code to use this interface rather than implementing direct integration with Cloud Pub/Sub.

Objectives

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

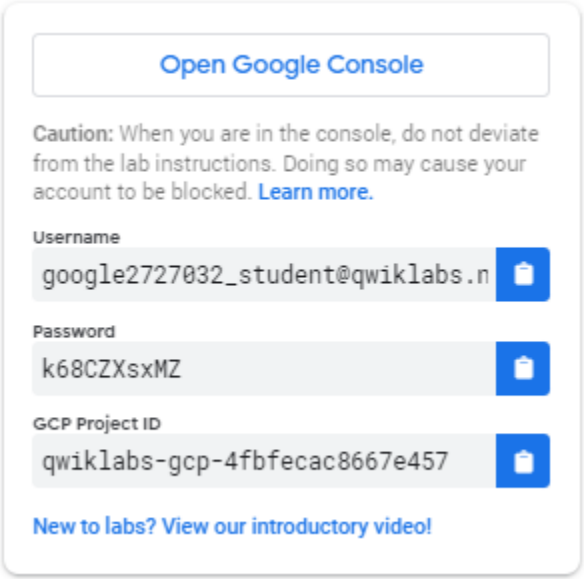
- Add Spring Integration Core to an application
- Create an outbound message gateway in your application
- Configure an application to publish messages through a gateway
- Bind the output channel of a message gateway to Cloud Pub/Sub

Task 0. Lab Preparation

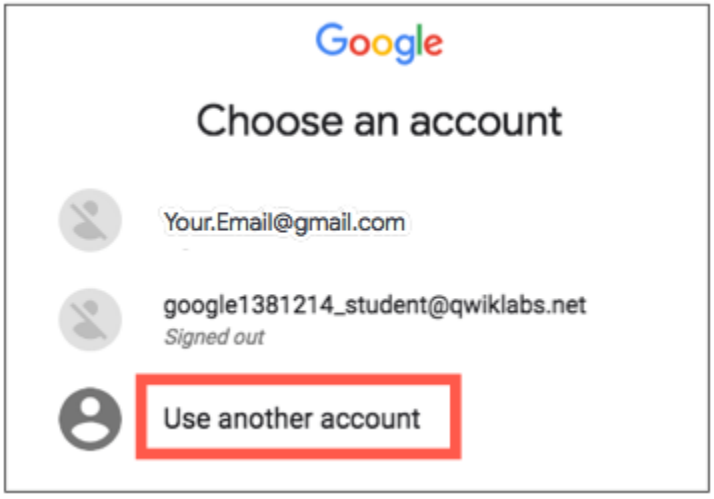
Access Qwiklabs

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**.



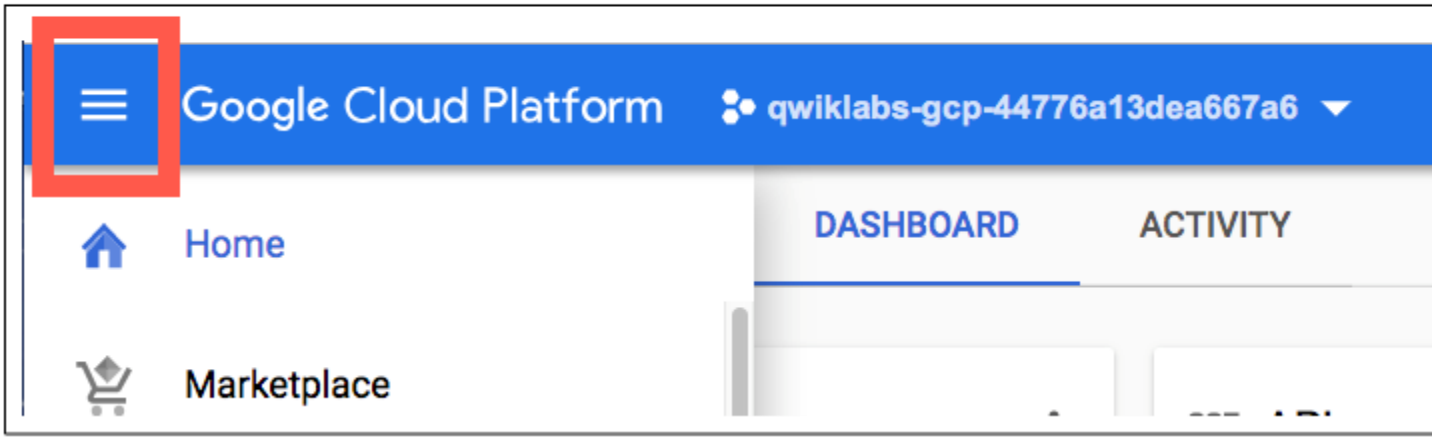
4. The Sign in page opens. Paste the username that you copied from the Connection Details panel. Then copy and paste the password.

   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:

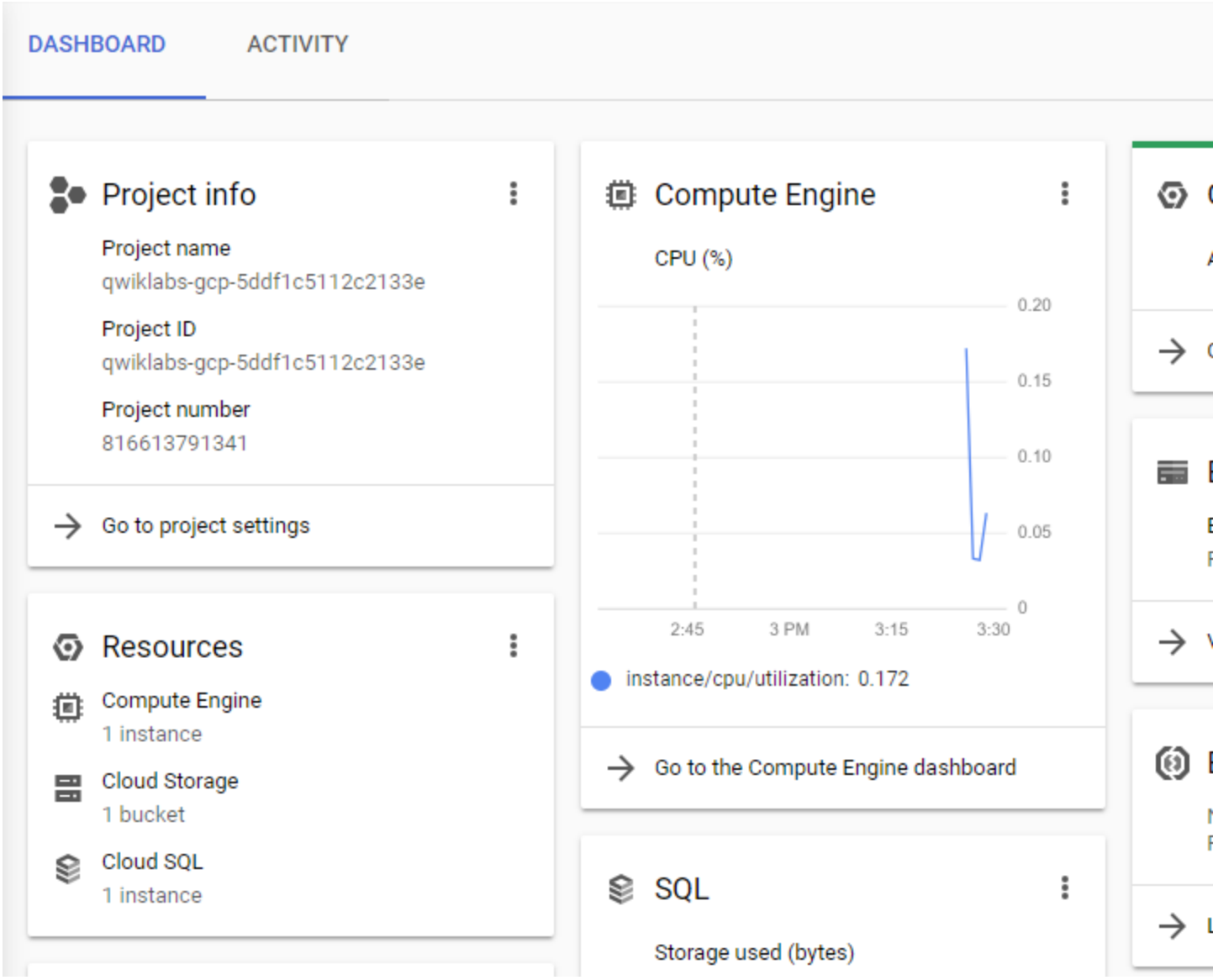
- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

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.

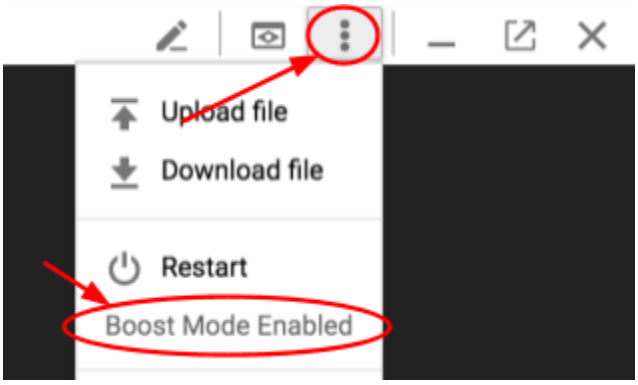


Fetch the application source files

The lab setup includes automated deployment of the services that you configured yourself in previous labs. When the setup is complete, copies of the demo application (configured so that they are ready for this lab session) are put into a Cloud Storage bucket named using the project ID for this lab.

Before you proceed with the tasks for this lab, you must first copy the demo application into Cloud Shell so you can continue to work on it.

1. In the upper-right corner of the screen, click **Activate Cloud Shell** ( ) to open Cloud Shell.
2. Click **Start Cloud Shell**.
3. If **Boost Mode Enabled** is not available (bold), enable boost mode for Cloud Shell.



4. 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)')
```

5. Verify that the demo application files were created.

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

Repeat the last step if the command reports an error or if it does not list the two folders for the `guestbook-frontend` application and the `guestbook-service` backend application.

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.

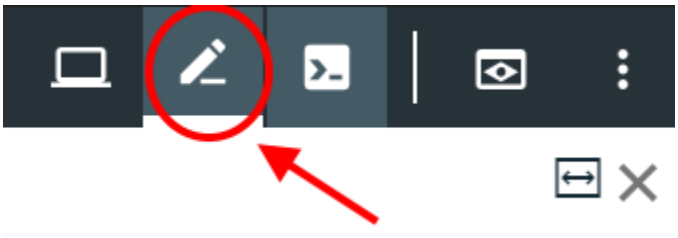
6. Copy the application folders to Cloud Shell.

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

7. Make the Maven wrapper scripts executable.

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

8. Click the pencil icon to open the Cloud Shell code editor.



Task 1. Add the Spring Integration core

Spring Integration core provides a framework for you to add a message gateway interface that can abstract from the underlying messaging system used.

In this task, you add the Spring Cloud Integration starter to the frontend application so that you can refactor the code to use a messaging gateway interface instead of using direct integration with Cloud Pub/Sub.

1. In the Cloud Shell code editor, open `~/guestbook-frontend/pom.xml`.
2. Insert the following new dependency at the end of the `<dependencies>` section, just before the closing `</dependencies>` tag:

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-core</artifactId>
</dependency>
```

Note
This dependency is in the standalone `<dependencies>` section, not in the `dependencyManagement` section.

Task 2. Create an outbound message gateway

In this task, you create an `OutboundGateway.java` file in the frontend application. The file contains a single method to send a text message.

1. In the Cloud Shell code editor, create a file named `OutboundGateway.java` in the `~/guestbook-frontend/src/main/java/com/example/frontend` directory.
2. Open `~/guestbook-frontend/src/main/java/com/example/frontend/OutboundGateway.java`.
3. Add the following code to the new file:

```
package com.example.frontend;

import org.springframework.integration.annotation.MessagingGateway;

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

Task 3. Publish the message

In this task, you modify the application to publish the message with the `FrontendController.post` method. This method enables you to use `OutboundGateway` to publish messages. Whenever someone posts a new guestbook message, `OutboundGateway` also sends it to a messaging system. At this point, the application does not know what messaging system is being used.

1. In the Cloud Shell code editor, open ~/`guestbook-frontend/src/main/java/com/example/frontend/FrontendController.java`.

2. Replace the code that references `pubSubTemplate` with references to `outboundGateway`:

Replace these lines:

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

With these lines:

```
    @Autowired
    private OutboundGateway outboundGateway;
```

Replace this line:

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

With this line:

```
    outboundGateway.publishMessage(name + ": " + message);
```

`FrontendController.java` should now look like the screenshot:

```java
package com.example.frontend;

import org.springframework.stereotype.Controller;
import org.springframework.web.client.RestTemplate;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.*;
import java.util.*;
import org.springframework.cloud.gcp.pubsub.core.*;

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

    @Autowired
    private OutboundGateway outboundGateway;

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

    @GetMapping("/")
    public String index(Model model) {
        if (model.containsAttribute("name")) {
            String name = (String) model.asMap().get("name");
            model.addAttribute("greeting", String.format("%s %s", greeting, name));
        }
        model.addAttribute("messages", client.getMessages().getContent());
        return "index";
    }

    @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
            Map<String, String> payload = new HashMap<>();
            payload.put("name", name);
            payload.put("message", message);
            client.add(payload);

            outboundGateway.publishMessage(name + ": " + message);
        }
        return "redirect:/";
    }
}
```

Task 4. Bind the output channel to the Cloud Pub/Sub topic

In the outbound gateway, you specified `messagesOutputChannel` as the default request channel. To define that channel to send the message to the Cloud Pub/Sub topic, you must create a new bean for that action in `FrontendApplication.java`.
In this task, you configure a service activator to bind `messagesOutputChannel` to use Cloud Pub/Sub.

1. In the Cloud Shell code editor, open ~/`guestbook-frontend/src/main/java/com/example/frontend/FrontendApplication.java`.
2. Add the following `import` directives below the existing `import` directives:

```
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.*;
```

3. Add the following code just before the closing brace at the end of the `FrontEndApplication` class definition:

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

`FrontendApplication.java` now looks like the following

```java
package com.example.frontend;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;
import org.springframework.hateoas.config.EnableHypermediaSupport;
import org.springframework.context.annotation.*;
import org.springframework.cloud.gcp.pubsub.core.*;
import org.springframework.cloud.gcp.pubsub.integration.outbound.*;
import org.springframework.integration.annotation.*;
import org.springframework.messaging.*;

@SpringBootApplication
// Enable consumption of HATEOS payloads
@EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HA
// Enable Feign Clients
@EnableFeignClients
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. Test the application in the Cloud Shell

In this task, you run the application in the Cloud Shell to test the new message gateway interface.

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

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

2. Run the backend service application.

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

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

```
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.

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

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

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

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

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

7. Open a new Cloud Shell session tab and check the Cloud Pub/Sub subscription for the published messages.

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

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

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.