

JAVAMS06

Integrating Pub/Sub with Spring

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 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 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 Pub/Sub


Setup and requirements


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.

[Open Google Console](#)

Caution: When you are in the console, do not deviate from the lab instructions. Doing so may cause your account to be blocked. [Learn more.](#)

Username
google2727032_student@qwiklabs.n 

Password
k68CZXsxMZ 

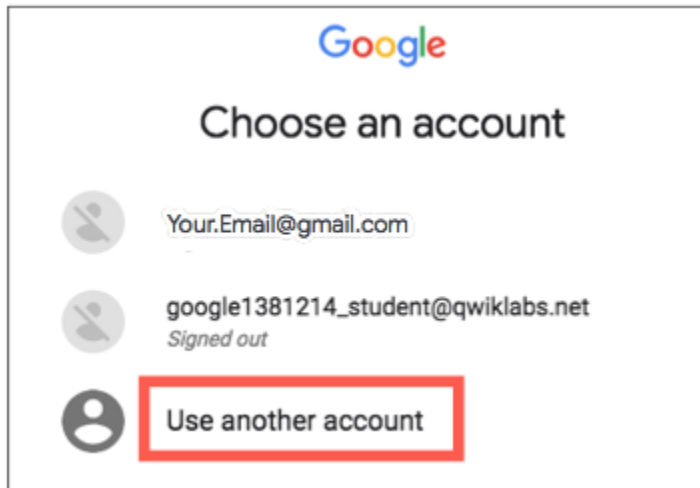
GCP Project ID
qwiklabs-gcp-4fbfecac8667e457 

[New to labs? View our introductory video!](#)

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.

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

3. On the Choose an account page, click **Use Another Account**. The Sign in page opens.



4. Paste the username that you copied from the Connection Details panel. Then copy and paste the password.

Note: You must use the credentials from the Connection Details panel. Do not use your Google Cloud Skills Boost credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

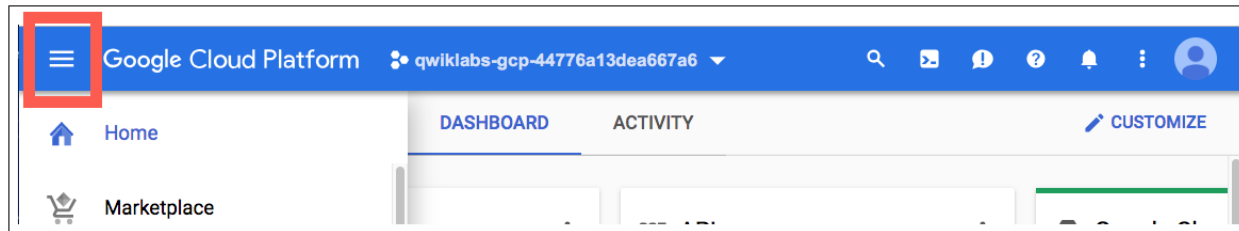
5. Click through the subsequent pages:

- 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.

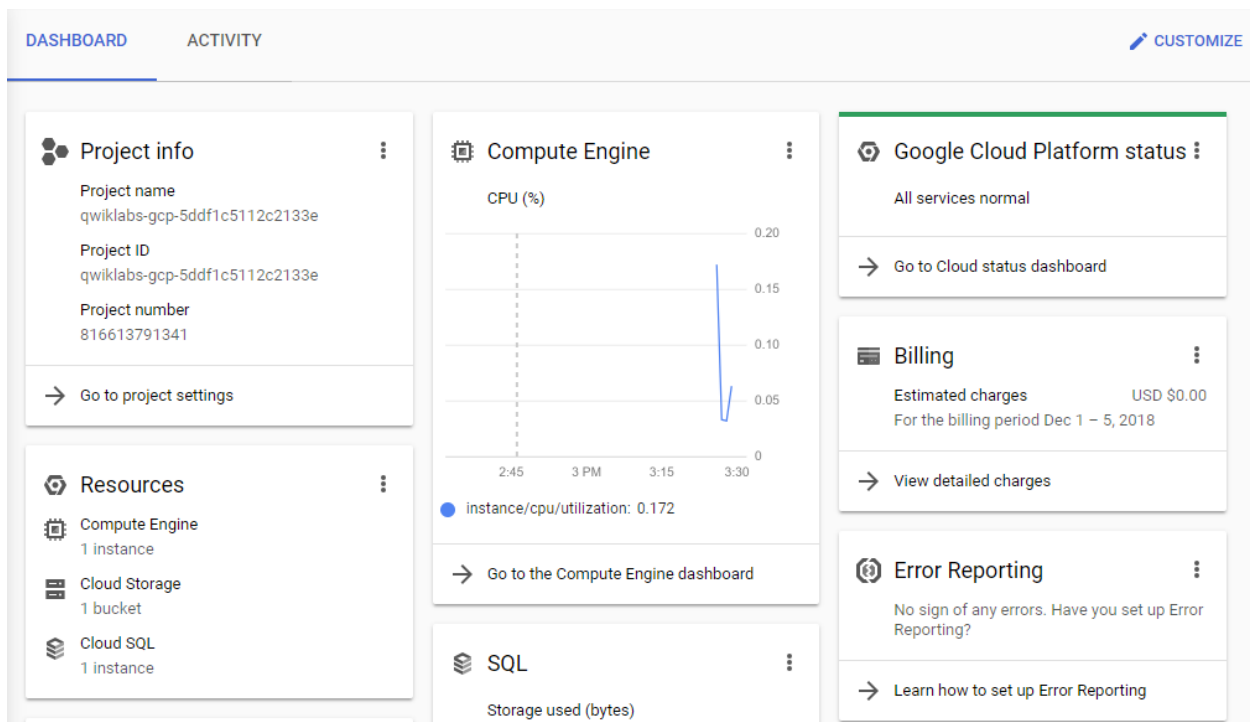
After a few moments, the Cloud console opens in this tab.

Note: You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-

left.



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



Task 1. Fetch the application source files

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

1. To begin the lab, click the **Activate Cloud Shell** button at the top of the Google Cloud Console and if prompted click **Continue**.
2. To activate the code editor, click the `Open Editor` button on the toolbar of the Cloud Shell window.
3. Click `Open in a new window` to set up the editor in a new tab with continued access to Cloud Shell.

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.

4. In Cloud Shell, click `Open Terminal` and then 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)')
```

Copied!

content_copy

5. Verify that the demo application files were created:

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

Copied!

content_copy

6. Copy the application folders to Cloud Shell:

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

Copied!

content_copy

7. Make the Maven wrapper scripts executable:

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

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

Copied!

content_copy

Now you're ready to go!

Task 2. Add the Spring Integration core

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 Pub/Sub.

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

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>
```

Copied!

content_copy

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

Task 3. 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);
}
```

Copied!

content_copy

Task 4. 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;
```

Copied!

content_copy

3. Replace this line:

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

with this line:

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

Copied!

content_copy

FrontendController.java should now look like the screenshot:

FrontendController.java x

```
1 package com.example.frontend;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.client.RestTemplate;
5 import org.springframework.ui.Model;
6 import org.springframework.web.bind.annotation.*;
7 import org.springframework.beans.factory.annotation.*;
8 import java.util.*;
9 // Add imports
10 import org.springframework.cloud.gcp.pubsub.core.*;
11
12
13 @Controller
14 @SessionAttributes("name")
15 public class FrontendController {
16     @Autowired
17     private GuestbookMessagesClient client;
18
19     @Value("${greeting:Hello}")
20     private String greeting;
21
22     @Autowired
23     private OutboundGateway outboundGateway;
24
25     @GetMapping("/")
26     public String index(Model model) {
27         if (model.containsKey("name")) {
28             String name = (String) model.asMap().get("name");
29             model.addAttribute("greeting", String.format("%s %s", greeting, name));
30         }
31         model.addAttribute("messages", client.getMessages().getContent());
32         return "index";
33     }
34
35     @PostMapping("/post")
36     public String post(@RequestParam String name, @RequestParam String message, Model model) {
37         model.addAttribute("name", name);
38         if (message != null && !message.trim().isEmpty()) {
39             // Post the message to the backend service
40             GuestbookMessage payload = new GuestbookMessage();
41             payload.setName(name);
42             payload.setMessage(message);
43             client.add(payload);
44
45             // At the very end, publish the message
46             outboundGateway.publishMessage(name + ": " + message);
47         }
48         return "redirect:/";
49     }
50 }
--
```

Task 5. Bind the output channel to the Pub/Sub topic

In this task, you configure a service activator to bind `messagesOutputChannel` to use Pub/Sub.

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

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.*;
```

Copied!

content_copy

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");
}
```

Copied!

content_copy

`FrontendApplication.java` now looks like the following:

FrontendApplication.java x

```
1  package com.example.frontend;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.boot.autoconfigure.SpringBootApplication;
5  import org.springframework.cloud.openfeign.EnableFeignClients;
6  import org.springframework.hateoas.config.EnableHypermediaSupport;
7  import org.springframework.context.annotation.*;
8  import org.springframework.cloud.gcp.pubsub.core.*;
9  import org.springframework.cloud.gcp.pubsub.integration.outbound.*;
10 import org.springframework.integration.annotation.*;
11 import org.springframework.messaging.*;
12
13 @SpringBootApplication
14 // Enable consumption of HATEOS payloads
15 @EnableHypermediaSupport(type = EnableHypermediaSupport.HypermediaType.HAL)
16 // Enable Feign Clients
17 @EnableFeignClients
18 public class FrontendApplication {
19
20     public static void main(String[] args) {
21         SpringApplication.run(FrontendApplication.class, args);
22     }
23
24     @Bean
25     @ServiceActivator(inputChannel = "messagesOutputChannel")
26     public MessageHandler messageSender(PubSubTemplate pubsubTemplate) {
27         return new PubSubMessageHandler(pubsubTemplate, "messages");
28     }
29 }
```

Task 6. Test the application in Cloud Shell

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

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

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

Copied!

content_copy

2. Run the backend service application:

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

Copied!

content_copy

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
```

Copied!

content_copy

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

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

Copied!

content_copy

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

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

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

Copied!

content_copy

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

Review

In this lab you added Spring Integration Core to an application. You also created an outbound message gateway in your application and configured an application to publish messages through a gateway. Finally, you bound the output channel of a message gateway to Pub/Sub