AVAMS05 Messaging with Pub/Sub

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 enhance your application to implement a message handling service with Pub/Sub so that it can publish a message to a topic that can then be subscribed and processed by other services.

Pub/Sub is a fully managed, real-time messaging service that enables you to send and receive messages between independent applications. Pub/Sub brings the scalability, flexibility, and reliability of enterprise message-oriented middleware to the cloud. By providing many-to-many, asynchronous messaging that decouples senders and receivers, Pub/Sub enables secure and highly available communication between independently written applications. Pub/Sub delivers low-latency, durable messaging that helps developers quickly integrate systems hosted on the Google Cloud and externally.

Objectives

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

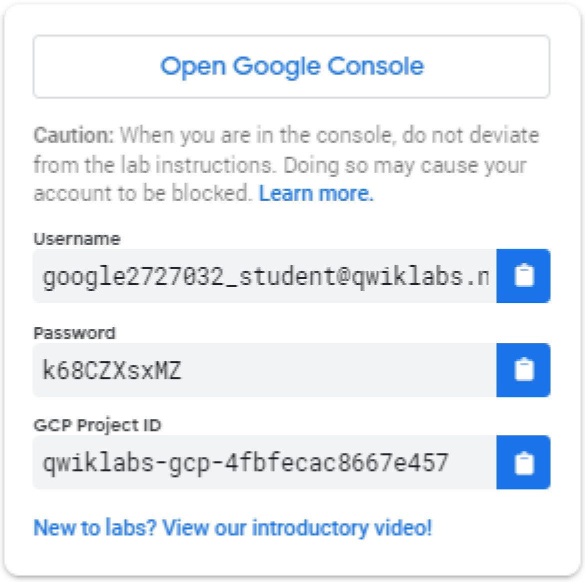
- Enable Pub/Sub and create a Pub/Sub topic

- Use Spring to add Pub/Sub support to your application
- Modify an application to publish Pub/Sub messages
- Create a Pub/Sub subscription
- Modify an application to process messages from a Pub/Sub subscription

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.



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

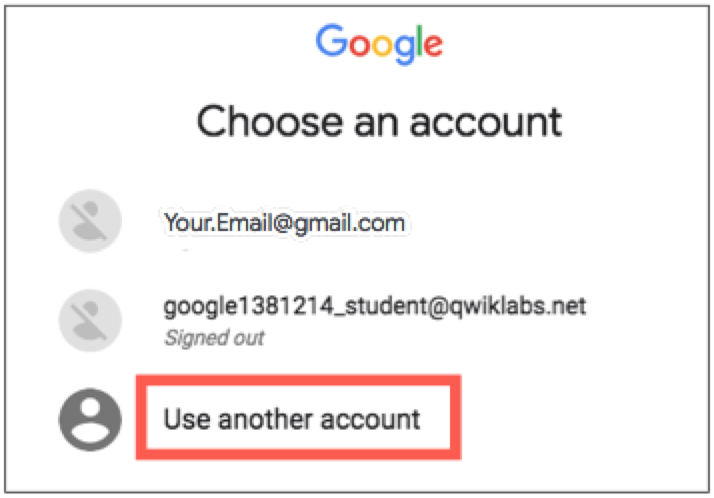
Password
k68CZXsxMZ

GCP Project ID
qwiklabs-gcp-4fbfecac8667e457

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.

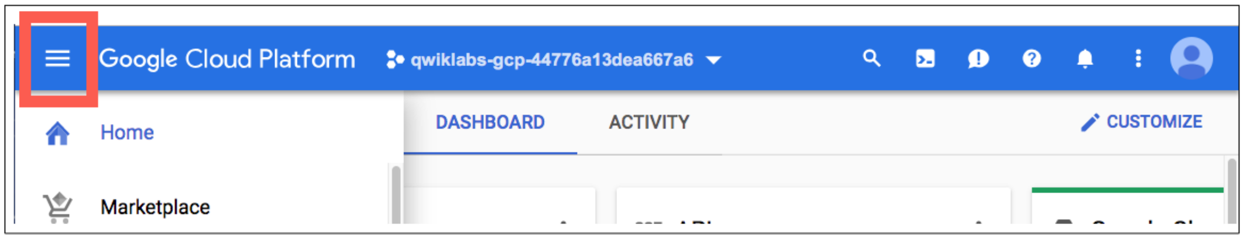
Note: You must use the credentials from the Connection Details panel. Do not use your Google Cloud Skills Boost credentials. If you have your own Google Cloud account, do not use it for this lab (avoids incurring charges).

5. Click through the subsequent pages:

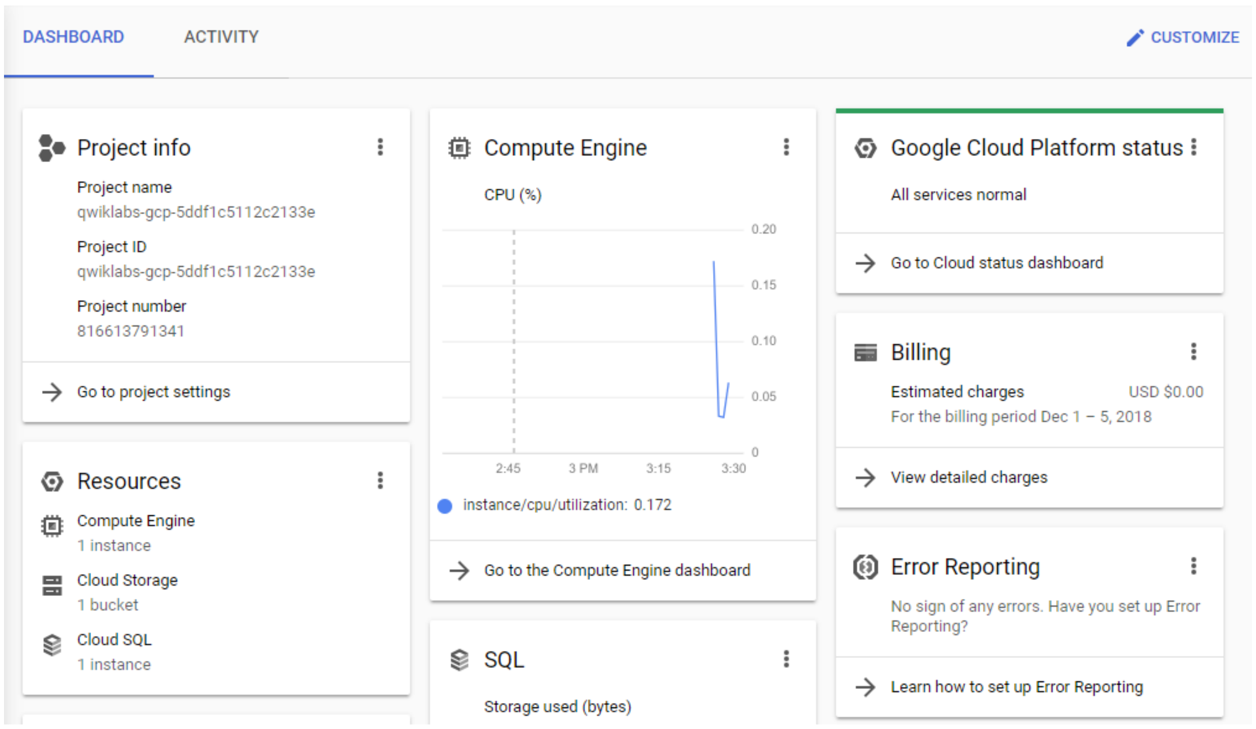
- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.

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.


DASHBOARD ACTIVITY CUSTOMIZE

Project info
Project name
qwiklabs-gcp-5ddf1c5112c2133e
Project ID
qwiklabs-gcp-5ddf1c5112c2133e
Project number
816613791341
Go to project settings

Compute Engine
CPU (%)
instance/cpu/utilization: 0.172
Go to the Compute Engine dashboard

Google Cloud Platform status
All services normal
Go to Cloud status dashboard

Billing
Estimated charges USD $0.00
For the billing period Dec 1 – 5, 2018
View detailed charges

Resources
Compute Engine
1 instance
Cloud Storage
1 bucket
Cloud SQL
1 instance

SQL
Storage used (bytes)

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

Task 1. Fetch the application source files

In this task you clone the 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 open the code editor in a separate tab.

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, 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
It may take a few minutes for provisioning to complete and the bucket to be created.

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. Enable Pub/Sub API

- In Cloud Shell, enable the Pub/Sub API:

```
gcloud services enable pubsub.googleapis.com
```

Copied!
content_copy

Task 3. Create a Pub/Sub topic

In this task, you create a Pub/Sub topic to which you will send a message.

- Use `gcloud` to create a Pub/Sub topic:

```
gcloud pubsub topics create messages
```
Copied!
content_copy

Task 4. Add Spring Cloud GCP Pub/Sub starter

In this task, you update the guestbook frontend application's `pom.xml` file to include the Spring Cloud GCP starter for Pub/Sub in the dependency section.

1. Open the Cloud Shell code editor.

Note: It is recommended to have files automatically save when you update them. Select **File > Auto Save** in the code editor menu.

2. In the code editor, open `~/guestbook-frontend/pom.xml`.

3. Insert the following new dependency at the end of the `<dependencies>` section, just before the closing `</dependencies>` tag:

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

Task 5. Publish a message

In this task, you use the `PubSubTemplate` bean in Spring Cloud GCP to publish a message to Pub/Sub. This bean is automatically configured and made available by the starter. You add `PubSubTemplate` to `FrontendController`.

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

2. Add the following statement immediately after the existing `import` directives:

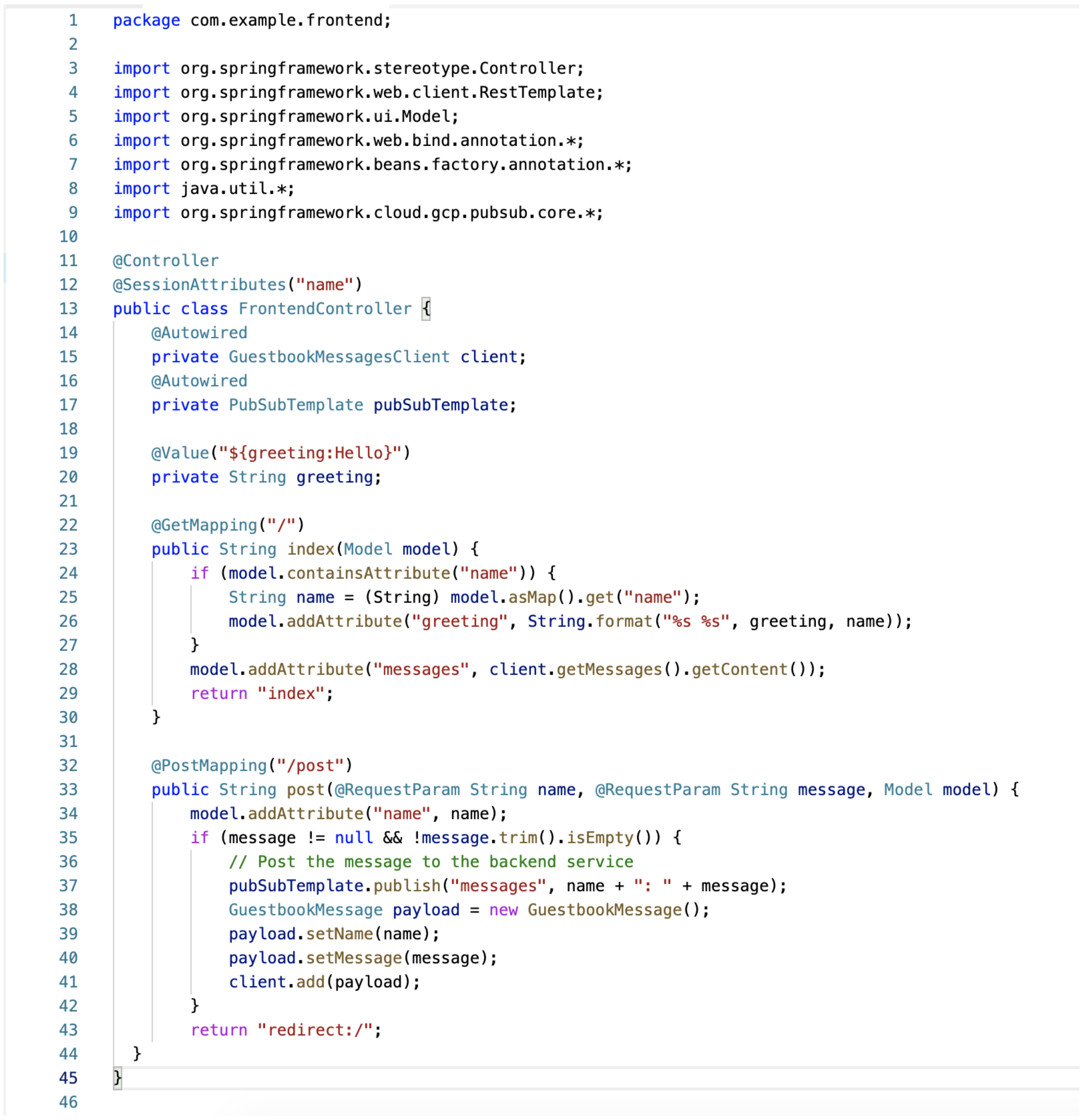
```
import org.springframework.cloud.gcp.pubsub.core.*;
```
Copied!
content_copy

3. Insert the following statement between the lines `private GuestbookMessagesClient client;` and `@Value("${greeting:Hello}")`:

```
    @Autowired
    private PubSubTemplate pubSubTemplate;
```
Copied!
content_copy

4. Add the following statement inside the if statement to process messages that aren't null or empty, just below the comment `// Post the message to the backend service`:

```
pubSubTemplate.publish("messages", name + ": " + message);
```
Copied!
content_copy
The code for `FrontendController.java` should now look like this 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 PubSubTemplate pubSubTemplate;

    @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
            pubSubTemplate.publish("messages", name + ": " + message);
            GuestbookMessage payload = new GuestbookMessage();
            payload.setName(name);
            payload.setMessage(message);
            client.add(payload);
        }
        return "redirect:/";
  }
}
```

Task 6. Test the application in the Cloud Shell

In this task, you run the application in the Cloud Shell to test the new Pub/Sub message handling code.

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.

The frontend application tries to publish a message to the Pub/Sub topic. You will check if this was successful in the next task.

Task 7. Create a subscription

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

Pub/Sub messages are delivered "at least once." Thus, you must deal with idempotence and you must deduplicate messages if you cannot process the same message more than once.

In this task, you create a Pub/Sub subscription and then test it by pulling messages from the subscription before and after using the frontend application to post a message.

1. Open a new Cloud Shell tab.

2. Create a Pub/Sub subscription:

```
gcloud pubsub subscriptions create messages-subscription-1 \
  --topic=messages
```
Copied!
content_copy

3. Pull messages from the subscription:

```
gcloud pubsub subscriptions pull messages-subscription-1
```
Copied!
content_copy
The `pull messages` command should report 0 items.

The message you posted earlier does not appear, because the message was published before the subscription was created.

4. Return to the frontend application, post another message, and then pull the message again:

```
gcloud pubsub subscriptions pull messages-subscription-1
```
Copied!

content_copy
The message appears. The message remains in the subscription until it is acknowledged.

5. Pull the message again and remove it from the subscription by using the auto-acknowledgement switch at the command line:

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

Copied!
content_copy

Task 8. Process messages in subscriptions

In this task, you use the Spring `PubSubTemplate` to listen to subscriptions.

1. In Cloud Shell, generate a new project from Spring Initializr:

```
cd ~
curl https://start.spring.io/starter.tgz \
  -d dependencies=web,cloud-gcp-pubsub \
  -d bootVersion=2.4.6.RELEASE \
  -d baseDir=message-processor | tar -xzvf -
```

Copied!
content_copy
This command generates a new Spring Boot project with the Pub/Sub starter preconfigured. The command also automatically downloads and unpacks the project into the `message-processor` directory structure.

2. Open `~/message-processor/pom.xml` to verify that the starter dependencies were automatically added.

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

3. To write the code to listen for new messages delivered to the topic, open `~/message-processor/src/main/java/com/example/demo/DemoApplication.java` in the Cloud Shell code editor.

4. Add the following `import` directives below the existing `import` directives:

```
import org.springframework.context.annotation.Bean;
import org.springframework.boot.ApplicationRunner;
import com.google.cloud.spring.pubsub.core.*;
```

Copied!
content_copy

5. Add the following code block to the class definition for `DemoApplication`, just above the existing definition for the main method:

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

Copied!
content_copy
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.

6. Add this line to change the port on `message-processor/src/main/resources/application.properties`:

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

Copied!
content_copy

7. Return to the Cloud Shell tab for the message processor to listen to the topic:

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

Copied!
content_copy

8. Open the browser with the frontend application, and post a few messages.
9. Verify that the Pub/Sub messages are received in the message processor.

The new messages should be displayed in the Cloud Shell tab where the message processor is running, as in the following example:

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

Task 9. Review

In this lab you enabled Pub/Sub and created a Pub/Sub topic. You used Spring to add Pub/Sub support to your application. You also modifed an application to publish Pub/Sub messages, and created a Pub/Sub subscription. Finally, you modifed an application to process messages from a Pub/Sub subscription.