Messaging with Google Cloud Pub/Sub

This guide walks you through the process of exchanging messages between different parts of a program, or different programs, using Spring Integration channel adapters and Google Cloud Pub/Sub as the underlying message exchange mechanism.

What you'll build

A Spring Boot web application that sends messages to itself and processes those messages.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
- A Google Cloud Platform project with billing and Pub/Sub enabled
- Google Cloud SDK

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Build with Gradle.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-messaging-gcp-pubsub.git`
- cd into `gs-messaging-gcp-pubsub/initial`
- Jump ahead to Add required dependencies.

When you finish, you can check your results against the code in `gs-messaging-gcp-pubsub/complete`.

Build with Gradle

Build with Maven

Build with your IDE

Add required dependencies

Add the following to your `pom.xml` file if you're using Maven:

COPY

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

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

Or, if you're using Gradle:

COPY

```
dependencies {
    ...
    compile("org.springframework.cloud:spring-cloud-gcp-starter-
pubsub:1.2.5.RELEASE")
    compile("org.springframework.integration:spring-integration-core")
    ...
}
```

If you're using Maven, you are also strongly encouraged to use the Spring Cloud GCP bill of materials to control the versions of your dependencies:

COPY

```
<properties>
    ...
    <spring-cloud-gcp.version>1.2.5.RELEASE</spring-cloud-gcp.version>
    ...
</properties>

<dependencyManagement>
    <dependencies>
       ...
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-gcp-dependencies</artifactId>
            <version>${spring-cloud-gcp.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        ...
    </dependencies>
</dependencyManagement>
```

Set up Google Cloud Pub/Sub environment

You will need a topic and a subscription to send and receive messages from Google Cloud Pub/Sub. You can create them in the Google Cloud Console or, programatically, with the `PubSubAdmin` class.

For this exercise, create a topic called "testTopic" and a subscription for that topic called "testSubscription".

Create application files

You'll need a class to include the channel adapter and messaging configuration. Create a PubSubApplication class with the @SpringBootApplication header, as is typical with a Spring Boot application.

`src/main/java/hello/PubSubApplication.java`

COPY

```
@SpringBootApplication
public class PubSubApplication {

  public static void main(String[] args) throws IOException {
    SpringApplication.run(PubSubApplication.class, args);
  }

}
```

Additionally, since you're building a web application, create a WebAppController class to separate between the controller and configuration logic.

`src/main/java/hello/WebAppController.java`

COPY

```
@RestController
public class WebAppController {
}
```

We're still missing two files for HTML and properties.

`src/main/resources/static/index.html`

COPY

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Spring Integration GCP sample</title>
</head>
<body>
<div name="formDiv">
  <form action="/publishMessage" method="post">
```

```
    Publish message: <input type="text" name="message" /> <input
type="submit" value="Publish!"/>
  </form>
</div>
</body>
</html>
```

`src/main/resources/application.properties`

```
#spring.cloud.gcp.project-id=[YOUR_GCP_PROJECT_ID_HERE]
#spring.cloud.gcp.credentials.location=file:[LOCAL_FS_CREDENTIALS_PATH]
```

The Spring Cloud GCP Core Boot starter can auto-configure these two properties and make them optional. Properties from the properties file always have precedence over the Spring Boot configuration. The Spring Cloud GCP Core Boot starter is bundled with the Spring Cloud GCP Pub/Sub Boot starter.

The GCP project ID is auto-configured from the `GOOGLE_CLOUD_PROJECT` environment variable, among several other sources. The OAuth2 credentials are auto-configured from the GOOGLE_APPLICATION_CREDENTIALS environment variable. If the Google Cloud SDK is installed, this environment variable is easily configured by running the `gcloud auth application-default login` command in the same process of the app, or a parent one.

Create an inbound channel adapter

An inbound channel adapter listens to messages from a Google Cloud Pub/Sub subscription and sends them to a Spring channel in an application.

Instantiating an inbound channel adapter requires a `PubSubTemplate` instance and the name of an existing subscription. `PubSubTemplate` is Spring's abstraction to subscribe to Google Cloud Pub/Sub topics. The Spring Cloud GCP Pub/Sub Boot starter provides an auto-configured `PubSubTemplate` instance which you can simply inject as a method argument.

`src/main/java/hello/PubSubApplication.java`

```
  @Bean
  public PubSubInboundChannelAdapter messageChannelAdapter(
    @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
    PubSubTemplate pubSubTemplate) {
```

```
  PubSubInboundChannelAdapter adapter =
    new PubSubInboundChannelAdapter(pubSubTemplate, "testSubscription");
  adapter.setOutputChannel(inputChannel);
  adapter.setAckMode(AckMode.MANUAL);

  return adapter;
  }
```

The message acknowledgement mode is set in the adapter to automatic, by default. This behaviour may be overridden, as shown in the example.

After the channel adapter is instantiated, an output channel where the adapter sends the received messages to must be configured.

`src/main/java/hello/PubSubApplication.java`

COPY

```
  @Bean
  public MessageChannel pubsubInputChannel() {
  return new DirectChannel();
  }
```

Attached to an inbound channel is a service activator which is used to process incoming messages.

`src/main/java/hello/PubSubApplication.java`

COPY

```
  @Bean
  @ServiceActivator(inputChannel = "pubsubInputChannel")
  public MessageHandler messageReceiver() {
  return message -> {
    LOGGER.info("Message arrived! Payload: " + new String((byte[])
 message.getPayload()));
    BasicAcknowledgeablePubsubMessage originalMessage =
    message.getHeaders().get(GcpPubSubHeaders.ORIGINAL_MESSAGE,
 BasicAcknowledgeablePubsubMessage.class);
    originalMessage.ack();
  };
  }
```

The `ServiceActivator` input channel name (e.g., `"pubsubInputChannel"`) must match the input channel method name. Whenever a new message arrives to that channel, it is processed by the returned `MessageHandler`.

In this example, the message is processed simply by logging its body and acknowledging it. In manual acknowledgement, a message is acknowledged using the `BasicAcknowledgeablePubsubMessage` object, which is attached to the `Message` headers and can be extracted using the `GcpPubSubHeaders.ORIGINAL_MESSAGE` key.

Create an outbound channel adapter

An outbound channel adapter listens to new messages from a Spring channel and publishes them to a Google Cloud Pub/Sub topic.

Instantiating an outbound channel adapter requires a `PubSubTemplate` and the name of an existing topic. `PubSubTemplate` is Spring's abstraction to publish messages to Google Cloud Pub/Sub topics. The Spring Cloud GCP Pub/Sub Boot starter provides an auto-configured `PubSubTemplate` instance.

`src/main/java/hello/PubSubApplication.java`

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

You can use a `MessageGateway` to write messages to a channel and publish them to Google Cloud Pub/Sub.

`src/main/java/hello/PubSubApplication.java`

```
@MessagingGateway(defaultRequestChannel = "pubsubOutputChannel")
public interface PubsubOutboundGateway {

void sendToPubsub(String text);
}
```
COPY

From this code, Spring auto-generates an object that can then be autowired into a private field in the application.

`src/main/java/hello/WebAppController.java`

```
  @Autowired
  private PubsubOutboundGateway messagingGateway;
```
COPY

Add controller logic

Add logic to your controller that lets you write to a Spring channel:

`src/main/java/hello/WebAppController.java`

```
package hello;

import hello.PubSubApplication.PubsubOutboundGateway;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.view.RedirectView;

@RestController
public class WebAppController {

  // tag::autowireGateway[]
  @Autowired
  private PubsubOutboundGateway messagingGateway;
  // end::autowireGateway[]

  @PostMapping("/publishMessage")
  public RedirectView publishMessage(@RequestParam("message") String
message) {
  messagingGateway.sendToPubsub(message);
  return new RedirectView("/");
  }
}
```
COPY

Authentication

Your application must be authenticated either via the GOOGLE_APPLICATION_CREDENTIALS environment variable or the `spring.cloud.gcp.credentials.location` property.

If you have the Google Cloud SDK installed, you can log in with your user account using the `gcloud auth application-default login` command.

Alternatively, you can download a service account credentials file from the Google Cloud Console and point the `spring.cloud.gcp.credentials.location` property in the `application.properties` file to it.

As a Spring Resource, the `spring.cloud.gcp.credentials.location` can also be obtained from places other than the file system, like a URL, classpath, etc.

Make the application executable

Although it is possible to package this service as a traditional WAR file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a Java `main()` method. Also, you use Spring's support for embedding the Tomcat servlet container as the HTTP runtime, instead of deploying to an external instance.

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `hello` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-messaging-gcp-pubsub-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-messaging-gcp-pubsub-0.1.0.jar
```

The steps described here create a runnable JAR. You can also build a classic WAR file.

Logging output is displayed. The service should be up and running within a few seconds.

Test the application

Now that the application is running, you can test it. Open http://localhost:8080, type a message in the input text box, press the "Publish!" button and verify that the message was correctly logged in your process terminal window.

Summary

Congratulations! You've just developed a Spring application that exchanges messages using Spring Integration GCP Pub/Sub channel adapters!

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.