Accediendo a Pub/Sub con Java

Elaborada por: M. en C. Ukranio Coronilla

Nota: Es importante que se encuentre activo el servicio Pub/Sub de la práctica anterior, así como la subscripción para el desarrollo de la presente práctica.

Ahora que hemos visto el funcionamiento del servicio Pub/Sub con ayuda de gcloud en las instancias de Google, vamos a implementar nuestros programas en Java que interactúen con dicho servicio.

A diferencia de librerías como Lanterna, la librería **Google Cloud Pub/Sub** tiene una gran cantidad de dependencias que a su vez requieren otras dependencias en las versiones adecuadas. Por esa razón utilizaremos la herramienta *Apache Maven* que se encargará de descargar todas las librerías necesarias para nuestro proyecto.

Maven

Vamos a instalar Maven en una instancia de Google con el siguiente procedimiento. Para instalar Maven es necesario que tengamos ya instalado el JDK de Java por lo que ejecutamos en la terminal SSH:

```
sudo apt update
```

```
sudo apt-get install openjdk-17-jdk
```

Posteriormente instalamos Maven con:

```
sudo apt install maven
```

Y una vez instalada debería mostrar la versión de Maven junto con la versión de Java preinstalada con:

```
mvn -version
```

En el siguiente link viene una descripción completa de cómo utilizar Maven por si queda alguna duda en la presente explicación:

https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

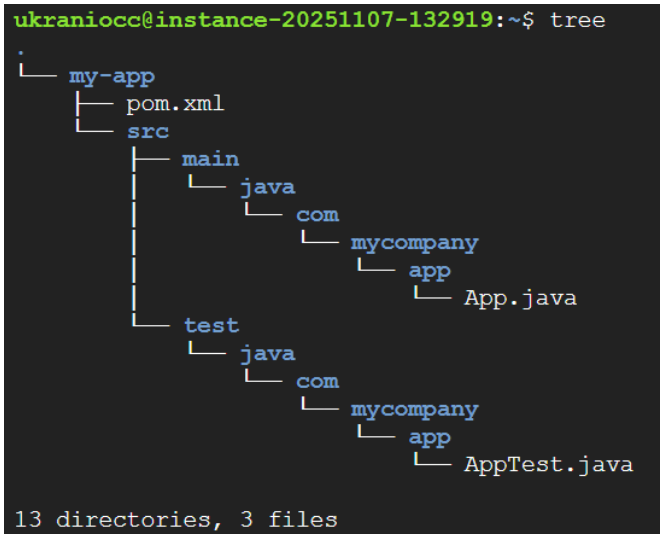
Para utilizar Maven se requiere crear un proyecto que va a albergar nuestro código, así como un código adicional para especificar sus dependencias. Para crear un proyecto básico utilizando una plantilla nos movemos a la carpeta donde deseamos ubicar al proyecto y dentro de la carpeta ejecutamos (al ser la primera vez que se ejecuta puede ser tardado descargar los repositorios y además puede requerir que se ejecute más de una vez si se queda bloqueado):

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -
DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.5 -
DinteractiveMode=false
```

Con esta instrucción se crea un proyecto Maven vacío generando tres archivos y doce directorios. Para visualizarlo instalamos el comando `tree` con:

```
sudo apt install tree
```

al ejecutarlo podemos ver el siguiente árbol de directorios:



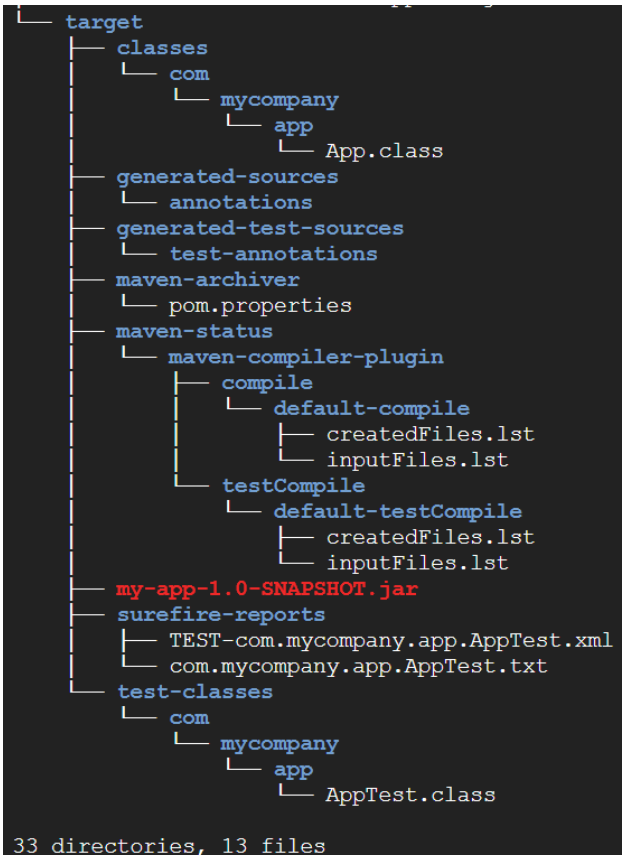
Observe que se ha creado el archivo **pom.xml** el cual es un archivo de configuración que contiene la información sobre las dependencias y plugins necesarios para construir un proyecto en Maven. En este archivo las dependencias se encuentran descritas entre las etiquetas `<dependencies>` y cada dependencia se encuentra entre las etiquetas `<dependency>` por ejemplo si abrimos el archivo pom.xml con un editor podemos ver que se encuentran dos dependencias de Junit, las cuales permiten hacer pruebas unitarias en el código:

```xml
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Optionally: parameterized tests support -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Para compilar el proyecto **nos ubicamos dentro de la carpeta `my-app`** y ejecutamos Maven con:

```
mvn clean package
```

La opción `clean` elimina archivos y directorios generados en una compilación anterior. Después de descargar las dependencias necesarias y compilar el código podemos observar que dentro de la carpeta `my-app` se ha creado la carpeta `target` la cual contiene varias carpetas con los archivos compilados y otros archivos adicionales:



Para ejecutar el proyecto tecleamos:

`java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App`

Observe que necesitamos además de la ubicación del archivo JAR la ruta completa de la clase que contiene el método `main` (`com.mycompany.app.App`). El método `main` se encuentra en el archivo `~/my-app/src/main/java/com/mycompany/app/App.java` y la ruta de la clase se toma a partir de la carpeta `~/my-app/src/main/java` donde Maven ubica por default el código fuente.

Cabe mencionar que en esta compilación no se ha creado un archivo JAR autocontenido pues si lo intentamos ejecutar como cualquier otro archivo JAR obtenemos:

```
ukraniocc@instance-20251107-132919:~/my-app$ java -jar target/my-app-1.0-SNAPSHOT.jar
no main manifest attribute, in target/my-app-1.0-SNAPSHOT.jar
```

Lo cual sucede porque java espera que dentro del archivo JAR esté definida en un archivo la ubicación de la función principal `main`.

Ahora que hemos creado un proyecto simple, le vamos a añadir el siguiente código de Java que publica mensajes (Pub):

```java
import com.google.cloud.pubsub.v1.Publisher;
import com.google.protobuf.ByteString;
import com.google.pubsub.v1.ProjectTopicName;
import com.google.pubsub.v1.PubsubMessage;
import java.util.concurrent.TimeUnit;

public class PublisherExample {
    public static void main(String... args) throws Exception {
        String projectId = "ID de tu proyecto";
        String topicId = "tema";

        ProjectTopicName topicName = ProjectTopicName.of(projectId, topicId);
        Publisher publisher = Publisher.newBuilder(topicName).build();

        try {
            String mensaje = "Publicando con Java. Mensaje 1.";
            ByteString data = ByteString.copyFromUtf8(mensaje);
            PubsubMessage pubsubMessage = PubsubMessage.newBuilder().setData(data).build();

            publisher.publish(pubsubMessage);
            System.out.println("Mensaje publicado: " + mensaje);

        } finally {
            publisher.shutdown();
            publisher.awaitTermination(1, TimeUnit.MINUTES);
        }
    }
}
```

Para esto nos vamos a la carpeta donde se encuentra el programa App.java que contiene la función principal:

```
cd /home/su_usuario/my-app/src/main/java/com/mycompany/app
```

Lo abrimos con el editor **pico** u otro de su preferencia y observamos el siguiente código de Hola mundo:

```java
package com.mycompany.app;

/**
 * Hello world!
 */
public class App {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Substituimos nuestro código manteniendo el nombre de la clase `App` así como la primera línea que indica la ubicación de la clase. La variable `projectId` la inicializamos con el identificador de nuestro proyecto (en mi caso `laboratorio-gcp-435316` verifique cual es el identificador de su proyecto) y la variable `topicId` con el tema que creamos en la práctica anterior, quedando el siguiente código:

```java
package com.mycompany.app;

import com.google.cloud.pubsub.v1.Publisher;
import com.google.protobuf.ByteString;
import com.google.pubsub.v1.ProjectTopicName;
import com.google.pubsub.v1.PubsubMessage;
import java.util.concurrent.TimeUnit;

public class App {
    public static void main(String... args) throws Exception {
        String projectId = "laboratorio-gcp-435316";
        String topicId = "new_transaction";

        ProjectTopicName topicName = ProjectTopicName.of(projectId, topicId);
        Publisher publisher = Publisher.newBuilder(topicName).build();

        try {
            String mensaje = "Publicando con Java mensaje 1";
            ByteString data = ByteString.copyFromUtf8(mensaje);
            PubsubMessage pubsubMessage = PubsubMessage.newBuilder().setData(data).build();

            publisher.publish(pubsubMessage);
            System.out.println("Mensaje publicado: " + mensaje);

        } finally {
            publisher.shutdown();
            publisher.awaitTermination(1, TimeUnit.MINUTES);
        }
    }
}
```

Ahora abrimos el archivo **pom.xml** y le agregamos la siguiente dependencia de Google Cloud Pub/Sub:

```xml
<dependency>
   <groupId>com.google.cloud</groupId>
   <artifactId>google-cloud-pubsub</artifactId>
   <version>1.123.6</version>
</dependency>
```

Con lo cual la sección de dependencias en el archivo pom.xml quedaría así:

```xml
<dependencies>

<dependency>
  <groupId>com.google.cloud</groupId>
  <artifactId>google-cloud-pubsub</artifactId>
  <version>1.123.6</version>
</dependency>

  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <scope>test</scope>
  </dependency>
  <!-- Optionally: parameterized tests support -->
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-params</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Ahora se procede a compilar con maven:

```
mvn clean package
```

Cuando la compilación tarda mucho podemos usar:

```
mvn clean package -T 1C -DskipTests -nsu
```

Con el cual se ocupan más núcleos para compilar, se omiten los test de prueba y no busca versiones nuevas en las dependencias.

Si lo intentamos ejecutar con:

```
java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
```

Obtendremos el siguiente error:

```
ukraniocc@instance-20251107-132919:~/my-app$ java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
Error: Unable to initialize main class com.mycompany.app.App
Caused by: java.lang.NoClassDefFoundError: com/google/pubsub/v1/TopicName
```

Esto sucede debido a que Maven solo empaqueta nuestro código, pero no el de las librerías externas de Pub/Sub. Para ejecutar este programa se utiliza un plugin de

Maven denominado `exec:java` el cual usa automáticamente todas las dependencias adicionales declaradas en el archivo pom.xml sin necesidad de empaquetarlas. Al usar este plugin el comando de ejecución queda como:

```
mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
```

Al ejecutarse correctamente imprime:

```
ukraniocc@instance-20251107-132919:~/my-app$ mvn exec:java -Dexec.mainClass="com.mycompany.app.App"
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< com.mycompany.app:my-app >----------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec-maven-plugin:3.6.2:java (default-cli) @ my-app ---
Mensaje publicado: Publicando con Java mensaje 1
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.121 s
[INFO] Finished at: 2025-11-10T18:47:28Z
[INFO] ------------------------------------------------------------------------
ukraniocc@instance-20251107-132919:~/my-app$
```

Después de ejecutar el código y enviar el mensaje, abrimos otra instancia y con ayuda de gcloud sacamos el mensaje del suscriptor con la instrucción que usamos la práctica pasada:

```
gcloud pubsub subscriptions pull new_transaction-sub --auto-ack
```

comprobando que el programa que hace la publicación funciona correctamente.

Advertencia: Tratar de empaquetar las librerías Pub/Sub junto con nuestro código en un único archivo JAR es muy tardado por ello se sugiere usar el plugin exec:java.

Ejercicio

Compile y ejecute el siguiente código que lee un mensaje del subscriptor probándolo con el programa java que acabamos de ejecutar. Adjunte la captura de pantalla donde se observan ambas terminales SSH con la ejecución correcta del Pub y del Sub.

```java
import com.google.cloud.pubsub.v1.AckReplyConsumer;
import com.google.cloud.pubsub.v1.MessageReceiver;
import com.google.cloud.pubsub.v1.Subscriber;
import com.google.pubsub.v1.ProjectSubscriptionName;
import com.google.pubsub.v1.PubsubMessage;

public class SubscriberExample {

    public static void main(String... args) throws Exception {
        String projectId = "ID de tu proyecto";
        String subscriptionId = "ID de la suscripción";

        ProjectSubscriptionName subscriptionName =
                ProjectSubscriptionName.of(projectId, subscriptionId);
```

```
        // Receptor que procesa cada mensaje entrante
        MessageReceiver receiver = (PubsubMessage message, AckReplyConsumer consumer) -> {
            String contenido = message.getData().toStringUtf8();
            System.out.println("Mensaje recibido: " + contenido);

            // Confirmar recepción para evitar reenvíos
            consumer.ack();
        };

        // Crear el subscriber
        Subscriber subscriber = Subscriber.newBuilder(subscriptionName, receiver).build();

        // Iniciar de forma asíncrona
        subscriber.startAsync().awaitRunning();
        System.out.println("Subscriber activo. Esperando mensajes (Ctrl + C para salir)...");

        // Manejo de apagado limpio al presionar Ctrl + C
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("\n Cerrando subscriber...");
            subscriber.stopAsync();
            System.out.println(" Subscriber detenido correctamente.");
        }));

        // Mantiene el hilo principal vivo mientras se reciben mensajes
        Thread.currentThread().join();
    }
}
```

Consideraciones finales

Pub/Sub es un canal de comunicación unidireccional, por lo tanto, en la arquitectura cliente-servidor se utilizan dos temas, uno para las solicitudes y otro para las respuestas. Dado que puede haber varios clientes y varios servidores, se utiliza un campo en los mensajes con un identificador que permita asociar el mensaje de respuesta con el de solicitud.