

Spring Cloud GCP Reference Documentation

1.0.0.M2

João André Martins , Jisha Abubaker , Ray Tsang , Mike Eltsufin , Artem Bilan

Copyright © 2017Pivotal Software, Inc.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

1. Introduction	1
2. Dependency Management	2
3. Spring Cloud GCP Core	3
3.1. Project ID	3
3.2. Credentials	3
Scopes	4
4. Spring Cloud GCP for Pub/Sub	5
4.1. Pub/Sub operations abstraction	5
Publishing to a topic	5
Subscribing to a subscription	6
Pulling messages from a subscription	6
4.2. Pub/Sub management	6
Creating a topic	6
Deleting a topic	6
Listing topics	7
Creating a subscription	7
Deleting a subscription	7
Listing subscriptions	7
4.3. Configuration	8
5. Spring Resources	9
5.1. Google Cloud Storage	9
5.2. Configuration	9
6. Spring JDBC	11
6.1. Prerequisites	11
6.2. Spring Boot Starter for Google Cloud SQL	11
DataSource creation flow	12
7. Spring Integration	13
7.1. Channel Adapters for Google Cloud Pub/Sub	13
Inbound channel adapter	13
Outbound channel adapter	14
7.2. Channel Adapters for Google Cloud Storage	14
Inbound channel adapter	15
Inbound streaming channel adapter	15
Outbound channel adapter	15
8. Spring Cloud Sleuth	17
8.1. Spring Boot Starter for Stackdriver Trace	17
9. Spring Cloud Config	19
9.1. Configuration	19
9.2. Quick start	20
9.3. Refreshing the configuration at runtime	20

1. Introduction

The Spring Cloud GCP project aims at making the Spring Framework a first-class citizen of Google Cloud Platform (GCP).

Currently, Spring Cloud GCP lets you leverage the power and simplicity of the Spring framework to:

1. Publish and subscribe from Google Cloud Pub/Sub topics
2. Configure Spring JDBC with a few properties to use Google Cloud SQL
3. Write and read from Spring Resources backed up by Google Cloud Storage
4. Exchange messages with Spring Integration using Google Cloud Pub/Sub on the background
5. Trace the execution of your app with Spring Cloud Sleuth and Google Stackdriver Trace
6. Configure your app with Spring Cloud Config, backed up by the Google Runtime Configuration API
7. Consume and produce Google Cloud Storage data via Spring Integration GCS Channel Adapters

2. Dependency Management

The Spring Cloud GCP Bill of Materials (BOM) contains the versions of all the dependencies it uses.

If you're a Maven user, adding the following to your pom.xml file will allow you to not specify any Spring Cloud GCP dependency versions. Instead, the version of the BOM you're using determines the versions of the used dependencies.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-gcp-dependencies</artifactId>
      <version>1.0.0.M2</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

In the following sections, it will be assumed you are using the Spring Cloud GCP BOM and the dependency snippets will not contain versions.

Gradle users can achieve the same kind of BOM experience using Spring's [dependency-management-plugin](#) Gradle plugin. For simplicity, the Gradle dependency snippets in the remainder of this document will also omit their versions.

3. Spring Cloud GCP Core

At the center of every Spring Cloud GCP module are the concepts of `GcpProjectIdProvider` and `CredentialsProvider`.

Spring Cloud GCP provides a Spring Boot starter to auto-configure the core components.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter'
}
```

3.1 Project ID

`GcpProjectIdProvider` is a functional interface that returns a GCP project ID string.

```
public interface GcpProjectIdProvider {
    String getProjectId();
}
```

The Spring Cloud GCP starter auto-configures a `GcpProjectIdProvider`. If a `spring.cloud.gcp.project-id` property is specified, the provided `GcpProjectIdProvider` returns that property value.

```
spring.cloud.gcp.project-id=my-gcp-project-id
```

Otherwise, the project ID is discovered based on a [set of rules](#):

1. The project ID specified by the `GOOGLE_CLOUD_PROJECT` environment variable
2. The Google App Engine project ID
3. The project ID specified in the JSON credentials file pointed by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
4. The Google Cloud SDK project ID
5. The Google Compute Engine project ID, from the Google Compute Engine Metadata Server

3.2 Credentials

`CredentialsProvider` is a functional interface that returns the credentials to authenticate and authorize calls to Google Cloud Client Libraries.

```
public interface CredentialsProvider {
    Credentials getCredentials() throws IOException;
}
```

The Spring Cloud GCP starter auto-configures a `CredentialsProvider`. It uses the `spring.cloud.gcp.credentials.location` property to locate the OAuth2 private key of a

Google service account. Keep in mind this property is a Spring Resource, so the credentials file can be obtained from a number of [different locations](#) such as the file system, classpath, URL, etc. The next example specifies the credentials location property in the file system.

```
spring.cloud.gcp.credentials.location=file:/usr/local/key.json
```

If that property isn't specified, the starter tries to discover credentials from a [number of places](#):

1. Credentials file pointed to by the `GOOGLE_APPLICATION_CREDENTIALS` environment variable
2. Credentials provided by the Google Cloud SDK `gcloud auth application-default login` command
3. Google App Engine built-in credentials
4. Google Cloud Shell built-in credentials
5. Google Compute Engine built-in credentials

Scopes

By default, the credentials provided by the Spring Cloud GCP Starter contain scopes for every service supported by Spring Cloud GCP.

Service	Scope
Pub/Sub	https://www.googleapis.com/auth/pubsub
Storage (Read Only)	https://www.googleapis.com/auth/devstorage.read_only
Storage (Write/Write)	https://www.googleapis.com/auth/devstorage.read_write
Runtime Config	https://www.googleapis.com/auth/cloudruntimeconfig
Trace (Append)	https://www.googleapis.com/auth/trace.append
Cloud Platform	https://www.googleapis.com/auth/cloud-platform

The Spring Cloud GCP starter allows you to configure a custom scope list for the provided credentials. To do that, specify a comma-delimited list of scopes in the `spring.cloud.gcp.credentials.scopes` property.

```
spring.cloud.gcp.credentials.scopes=https://www.googleapis.com/auth/pubsub,https://www.googleapis.com/auth/sqlservice.admin
```

You can also use `DEFAULT_SCOPES` placeholder as a scope to represent the starters default scopes, and append the additional scopes you need to add.

```
spring.cloud.gcp.credentials.scopes=DEFAULT_SCOPES,https://www.googleapis.com/auth/cloud-vision
```

4. Spring Cloud GCP for Pub/Sub

Spring Cloud GCP provides an abstraction layer to publish to and subscribe from Google Cloud Pub/Sub topics and to create, list or delete Google Cloud Pub/Sub topics and subscriptions.

A Spring Boot starter is provided to auto-configure the various required Pub/Sub components.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-pubsub'
}
```

4.1 Pub/Sub operations abstraction

`PubSubOperations` is an abstraction that allows Spring users to use Google Cloud Pub/Sub without depending on any Google Cloud Pub/Sub API semantics. It provides the common set of operations needed to interact with Google Cloud Pub/Sub. `PubSubTemplate` is the default implementation of `PubSubOperations` and it uses the [Google Cloud Java Client for Pub/Sub](#) to interact with Google Cloud Pub/Sub.

`PubSubTemplate` depends on a `PublisherFactory`, which is a functional interface to provide a Google Cloud Java Client for Pub/Sub `Publisher`. The Spring Boot starter for GCP Pub/Sub auto-configures a `PublisherFactory` with default settings and uses the `GcpProjectIdProvider` and `CredentialsProvider` auto-configured by the Spring Boot GCP starter.

The `PublisherFactory` implementation provided by Spring Cloud GCP Pub/Sub, `DefaultPublisherFactory`, caches `Publisher` instances by topic name, in order to optimize resource utilization.

Publishing to a topic

`PubSubTemplate` provides asynchronous methods to publish messages to a Google Cloud Pub/Sub topic. It supports different types of payloads, including `Strings` with different encodings, `byte[]`, `ByteString` and `PubsubMessage`.

```
public ListenableFuture<String> publish(String topic, String payload, Map<String, String> headers)

public ListenableFuture<String> publish(String topic, String payload, Map<String, String> headers,
    Charset charset)

public ListenableFuture<String> publish(String topic, byte[] payload, Map<String, String> headers)

public ListenableFuture<String> publish(String topic, ByteString payload, Map<String, String> headers)

public ListenableFuture<String> publish(String topic, PubsubMessage pubsubMessage)
```

Here is an example of how to publish a message to a Google Cloud Pub/Sub topic:

```
public void publishMessage() {
    this.pubSubTemplate.publish("topic", "your message payload", ImmutableMap.of("key1", "val1"));
}
```

Subscribing to a subscription

Google Cloud Pub/Sub allows many subscriptions to be associated to the same topic. `PubSubTemplate` allows you to subscribe to subscriptions via the `subscribe()` method. It relies on a `SubscriberFactory` object, whose only task is to generate Google Cloud Pub/Sub `Subscriber` objects. When subscribing to a subscription, messages will be pulled from Google Cloud Pub/Sub asynchronously, on a certain interval.

The Spring Boot starter for Google Cloud Pub/Sub auto-configures a `SubscriberFactory`.

Pulling messages from a subscription

Google Cloud Pub/Sub supports the synchronous pulling of messages from a subscription. This is different from subscribing to a subscription, in the sense that subscribing is an asynchronous task which polls the subscription on a set interval.

The `pullNext()` method allows for a single message to be pulled from a subscription. The `pull()` method pulls a number of messages from a subscription, allowing for the retry settings to be configured.

`PubSubTemplate` uses a special subscriber generated by its `SubscriberFactory` to pull messages.

4.2 Pub/Sub management

`PubSubAdmin` is the abstraction provided by Spring Cloud GCP to manage Google Cloud Pub/Sub resources. It allows for the creation, deletion and listing of topics and subscriptions.

`PubSubAdmin` depends on `GcpProjectIdProvider` and either a `CredentialsProvider` or a `TopicAdminClient` and a `SubscriptionAdminClient`. If given a `CredentialsProvider`, it creates a `TopicAdminClient` and a `SubscriptionAdminClient` with the Google Cloud Java Library for Pub/Sub default settings. The Spring Boot starter for GCP Pub/Sub auto-configures a `PubSubAdmin` object using the `GcpProjectIdProvider` and the `CredentialsProvider` auto-configured by the Spring Boot GCP Core starter.

Creating a topic

`PubSubAdmin` implements a method to create topics:

```
public Topic createTopic(String topicName)
```

Here is an example of how to create a Google Cloud Pub/Sub topic:

```
public void newTopic() {
    pubSubAdmin.createTopic("topicName");
}
```

Deleting a topic

`PubSubAdmin` implements a method to delete topics:

```
public void deleteTopic(String topicName)
```

Here is an example of how to delete a Google Cloud Pub/Sub topic:

```
public void deleteTopic() {
    pubSubAdmin.deleteTopic("topicName");
}
```


Listing topics

PubSubAdmin implements a method to list topics:

```
public List<Topic> listTopics
```

Here is an example of how to list every Google Cloud Pub/Sub topic name in a project:

```
public List<String> listTopics() {
    return pubSubAdmin
        .listTopics()
        .stream()
        .map(Topic::getNameAsTopicName)
        .map(TopicName::getTopic)
        .collect(Collectors.toList());
}
```

Creating a subscription

PubSubAdmin implements a method to create subscriptions to existing topics:

```
public Subscription createSubscription(String subscriptionName, String topicName, Integer ackDeadline,
    String pushEndpoint)
```

Here is an example of how to create a Google Cloud Pub/Sub subscription:

```
public void newSubscription() {
    pubSubAdmin.createSubscription("subscriptionName", "topicName", 10, "http://my.endpoint/push");
}
```

Alternative methods with default settings are provided for ease of use. The default value for `ackDeadline` is 10 seconds. If `pushEndpoint` isn't specified, the subscription uses message pulling, instead.

```
public Subscription createSubscription(String subscriptionName, String topicName)
```

```
public Subscription createSubscription(String subscriptionName, String topicName, Integer ackDeadline)
```

```
public Subscription createSubscription(String subscriptionName, String topicName, String pushEndpoint)
```

Deleting a subscription

PubSubAdmin implements a method to delete subscriptions:

```
public void deleteSubscription(String subscriptionName)
```

Here is an example of how to delete a Google Cloud Pub/Sub subscription:

```
public void deleteSubscription() {
    pubSubAdmin.deleteSubscription("subscriptionName");
}
```

Listing subscriptions

PubSubAdmin implements a method to list subscriptions:

```
public List<Subscription> listSubscriptions()
```

Here is an example of how to list every subscription name in a project:

```

public List<String> listSubscriptions() {
    return pubSubAdmin
        .listSubscriptions()
        .stream()
        .map(Subscription::getNameAsSubscriptionName)
        .map(SubscriptionName::getSubscription)
        .collect(Collectors.toList());
}

```

4.3 Configuration

The Spring Boot starter for Google Cloud Pub/Sub provides the following configuration options:

Name	Description	Optional	Default value
<code>spring.cloud.gcp.pubsub.enable-auto-config</code>	Enables or disables Pub/Sub auto-configuration	Yes	true
<code>spring.cloud.gcp.pubsub.subscriber-executor-threads</code>	Number of threads used by Subscriber instances created by SubscriberFactory	Yes	4
<code>spring.cloud.gcp.pubsub.publisher-executor-threads</code>	Number of threads used by Publisher instances created by PublisherFactory	Yes	4
<code>spring.cloud.gcp.pubsub.project-id</code>	GCP project ID where the Google Cloud Pub/Sub API is hosted, if different from the one in the Spring Cloud GCP Core Module	Yes	
<code>spring.cloud.gcp.pubsub.credentials.location</code>	Auth2 credentials for authenticating with the Google Cloud Pub/Sub API, if different from the ones in the Spring Cloud GCP Core Module	Yes	
<code>spring.cloud.gcp.pubsub.credentials.scopes</code>	Auth2 scopes for Spring Cloud GCP Config credentials	Yes	https://www.googleapis.com/auth/pubsub

5. Spring Resources

[Spring Resources](#) are an abstraction for a number of low-level resources, such as file system files, classpath files, servlet context-relative files, etc. Spring Cloud GCP adds a new resource type: a Google Cloud Storage (GCS) object.

A Spring Boot starter is provided to auto-configure the various Storage components.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-storage'
}
```

5.1 Google Cloud Storage

The Spring Resource Abstraction for Google Cloud Storage allows GCS objects to be accessed by their GCS URL:

```
@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;
```

This creates a `Resource` object that can be used to read the object, among [other possible operations](#).

It is also possible to write to a `Resource`, although a `WritableResource` is required.

```
@Value("gs://[YOUR_GCS_BUCKET]/[GCS_FILE_NAME]")
private Resource gcsResource;
...
try (OutputStream os = ((WritableResource) gcsResource).getOutputStream()) {
    os.write("foo".getBytes());
}
```

If the resource path refers to an object on Google Cloud Storage (as opposed to a bucket), then the resource can be cast as a `GoogleStorageResourceObject` and the `getGoogleStorageObject` method can be called to obtain a [Blob](#). This type represents a GCS file, which has associated [metadata](#), such as content-type, that can be set. The `createSignedUrl` method can also be used to obtain [signed URLs](#) for GCS objects. However, creating signed URLs requires that the resource was created using service account credentials.

The Spring Boot Starter for Google Cloud Storage auto-configures the `Storage` bean required by the `spring-cloud-gcp-storage` module, based on the `CredentialsProvider` provided by the Spring Boot GCP starter.

5.2 Configuration

The Spring Boot Starter for Google Cloud Storage provides the following configuration options:

Name	Description	Optional	Default value
------	-------------	----------	---------------

<code>spring.cloud.gcp.storage.create-files</code>	Creates files and buckets on Google Cloud Storage when writes are made to non-existent files	Yes	true
<code>spring.cloud.gcp.storage.credentials.location</code>	OAuth2 credentials for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	Yes	
<code>spring.cloud.gcp.storage.credentials.scopes</code>	OAuth2 scope for Spring Cloud GCP Config credentials	Yes	https://www.googleapis.com/auth/devstorage.read_write

6. Spring JDBC

Spring Cloud GCP adds integrations with [Spring JDBC](#) so you can run your MySQL or PostgreSQL databases in Google Cloud SQL using Spring JDBC, or other libraries that depend on it like Spring Data JPA.

The Cloud SQL support is provided by Spring Cloud GCP in the form of two Spring Boot starters, one for MySQL and another one for PostgreSQL. The role of the starters is to read configuration from properties and assume default settings so that user experience connecting to MySQL is as simple as possible.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-sql-mysql</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-gcp-starter-sql-postgresql</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-mysql'
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-sql-postgresql'
}
```

6.1 Prerequisites

In order to use the Spring Boot Starters for Google Cloud SQL, the Google Cloud SQL API must be enabled in your GCP project.

To do that, go to the [API library page](#) of the Google Cloud Console, search for "Cloud SQL API", click the first result and enable the API.

Note

There are several similar "Cloud SQL" results. You must access the "Google Cloud SQL API" one and enable the API from there.

6.2 Spring Boot Starter for Google Cloud SQL

The Spring Boot Starters for Google Cloud SQL provide an auto-configured [DataSource](#) object. Coupled with Spring JDBC, it provides a [JdbcTemplate](#) object bean that allows for operations such as querying and modifying a database.

```
public List<Map<String, Object>> listUsers() {
    return jdbcTemplate.queryForList("SELECT * FROM user;");
}
```

You can rely on [Spring Boot data source auto-configuration](#) to configure a `DataSource` bean. In other words, properties like the SQL username, `spring.datasource.username`, and password, `spring.datasource.password` can be used. There is also some configuration specific to Google Cloud SQL:

Property name	Description	Default value	Unused if specified property(ies)
<code>spring.cloud.gcp.sql.enabled</code>	Enables or disables Cloud SQL auto configuration	true	
<code>spring.cloud.gcp.sql.database-name</code>	Name of the database to connect to.		<code>spring.datasource.url</code>
<code>spring.cloud.gcp.sql.connection-name</code>	A string containing a Google Cloud SQL instance's project ID, region and name, each separated by a colon. For example, my-project-id:my-region:my-instance-name.		<code>spring.datasource.url</code>
<code>spring.cloud.gcp.sql.credentials-path</code>	File system path to the Google OAuth2 credentials private key file. Used to authenticate and authorize new connections to a Google Cloud SQL instance.	Default credentials provided by the Spring GCP Boot starter	

DataSource creation flow

Based on the previous properties, the Spring Boot starter for Google Cloud SQL creates a `CloudSqlJdbcInfoProvider` object which is used to obtain an instance's JDBC URL and driver class name. If you provide your own `CloudSqlJdbcInfoProvider` bean, it is used instead and the properties related to building the JDBC URL or driver class are ignored.

The `DataSourceProperties` object provided by Spring Boot Autoconfigure is mutated in order to use the JDBC URL and driver class names provided by `CloudSqlJdbcInfoProvider`, unless those values were provided in the properties. It is in the `DataSourceProperties` mutation step that the credentials factory is registered in a system property to be `SqlCredentialFactory`.

`DataSource` creation is delegated to [Spring Boot](#). You can select the type of connection pool (e.g., Tomcat, HikariCP, etc.) by [adding their dependency to the classpath](#).

Using the created `DataSource` in conjunction with Spring JDBC provides you with a fully configured and operational `JdbcTemplate` object that you can use to interact with your SQL database. You can connect to your database with as little as a database and instance names.

7. Spring Integration

Spring Cloud GCP provides Spring Integration adapters that allow your applications to use Enterprise Integration Patterns backed up by Google Cloud Platform services.

7.1 Channel Adapters for Google Cloud Pub/Sub

The channel adapters for Google Cloud Pub/Sub connect your Spring [MessageChannels](#) to Google Cloud Pub/Sub topics and subscriptions. This enables messaging between different processes, applications or micro-services backed up by Google Cloud Pub/Sub.

The Spring Integration Channel Adapters for Google Cloud Pub/Sub are included in the `spring-cloud-gcp-pubsub` module.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-pubsub'
}
```

Inbound channel adapter

`PubSubInboundChannelAdapter` is the inbound channel adapter for GCP Pub/Sub that listens to a GCP Pub/Sub subscription for new messages. It converts new messages to an internal Spring [Message](#) and then sends it to the bound output channel.

To use the inbound channel adapter, a `PubSubInboundChannelAdapter` must be provided and configured on the user application side.

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

@Bean
public PubSubInboundChannelAdapter messageChannelAdapter(
    @Qualifier("pubsubInputChannel") MessageChannel inputChannel,
    SubscriberFactory subscriberFactory) {
    PubSubInboundChannelAdapter adapter =
        new PubSubInboundChannelAdapter(subscriberFactory, "subscriptionName");
    adapter.setOutputChannel(inputChannel);
    adapter.setAckMode(AckMode.MANUAL);

    return adapter;
}
```

In the example, we first specify the `MessageChannel` where the adapter is going to write incoming messages to. The `MessageChannel` implementation isn't important here. Depending on your use case, you might want to use a `MessageChannel` other than `PublishSubscribeChannel`.

Then, we declare a `PubSubInboundChannelAdapter` bean. It requires the channel we just created and a `SubscriberFactory`, which creates `Subscriber` objects from the Google Cloud Java Client for Pub/Sub. The Spring Boot starter for GCP Pub/Sub provides a configured `SubscriberFactory`.

It is also possible to set the message acknowledgement mode on the adapter, which is automatic by default. On automatic acking, a message is acked with GCP Pub/Sub if the adapter sent it to the channel and no exceptions were thrown. If a `RuntimeException` is thrown while the message is processed, then the message is nacked. On manual acking, the adapter attaches an `AckReplyConsumer` object to the `Message` headers, which users can extract using the `GcpHeaders.ACKNOWLEDGEMENT` key and use to (n)ack a message.

```
@Bean
@ServiceActivator(inputChannel = "pubsubInputChannel")
public MessageHandler messageReceiver() {
    return message -> {
        LOGGER.info("Message arrived! Payload: " + message.getPayload());
        AckReplyConsumer consumer =
            message.getHeaders().get(GcpHeaders.ACKNOWLEDGEMENT, AckReplyConsumer.class);
        consumer.ack();
    };
}
```

Outbound channel adapter

`PubSubMessageHandler` is the outbound channel adapter for GCP Pub/Sub that listens for new messages on a Spring `MessageChannel`. It uses `PubSubTemplate` to convert new `Message` instances to the GCP Pub/Sub format and post them to a GCP Pub/Sub topic.

To use the outbound channel adapter, a `PubSubMessageHandler` bean must be provided and configured on the user application side.

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

The provided `PubSubTemplate` contains all the necessary configuration to publish messages to a GCP Pub/Sub topic.

`PubSubMessageHandler` publishes messages asynchronously by default. A publish timeout can be configured for synchronous publishing. If none is provided, the adapter waits indefinitely for a response.

It is possible to set user-defined callbacks for the `publish()` call in `PubSubMessageHandler` through the `setPublishFutureCallback()` method. These are useful to process the message ID, in case of success, or the error if any was thrown.

7.2 Channel Adapters for Google Cloud Storage

The channel adapters for Google Cloud Storage allow you to read and write files to Google Cloud Storage through `MessageChannels`.

Spring Cloud GCP provides two inbound adapters, `GcsInboundFileSynchronizingMessageSource` and `GcsStreamingMessageSource`, and one outbound adapter, `GcsMessageHandler`.

The Spring Integration Channel Adapters for Google Cloud Storage are included in the `spring-cloud-gcp-storage` module.

To use the Storage portion of Spring Integration for Spring Cloud GCP, you must also provide the `spring-integration-file` dependency, since they aren't pulled transitively.

Maven coordinates, using Spring Cloud GCP BOM:

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
</dependency>
<dependency>
  <groupId>com.google.cloud</groupId>
  <artifactId>google-cloud-storage</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.integration', name: 'spring-integration-file'
    compile group: 'com.google.cloud', name: 'google-cloud-storage'
}
```

Inbound channel adapter

The Google Cloud Storage inbound channel adapter polls a Google Cloud Storage bucket for new files and sends each of them in a Message payload to the MessageChannel specified in the @InboundChannelAdapter annotation. The files are temporarily stored in a folder in the local file system.

Here is an example of how to configure a Google Cloud Storage inbound channel adapter.

```
@Bean
@InboundChannelAdapter(channel = "new-file-channel", poller = @Poller(fixedDelay = "5000"))
public MessageSource<File> synchronizerAdapter(Storage gcs) {
    GcsInboundFileSynchronizer synchronizer = new GcsInboundFileSynchronizer(gcs);
    synchronizer.setRemoteDirectory("your-gcs-bucket");

    GcsInboundFileSynchronizingMessageSource synchAdapter =
        new GcsInboundFileSynchronizingMessageSource(synchronizer);
    synchAdapter.setLocalDirectory(new File("local-directory"));

    return synchAdapter;
}
```

Inbound streaming channel adapter

The inbound streaming channel adapter is similar to the normal inbound channel adapter, except it does not require files to be stored in the file system.

Here is an example of how to configure a Google Cloud Storage inbound streaming channel adapter.

```
@Bean
@InboundChannelAdapter(channel = "streaming-channel", poller = @Poller(fixedDelay = "5000"))
public MessageSource<InputStream> streamingAdapter(Storage gcs) {
    GcsStreamingMessageSource adapter =
        new GcsStreamingMessageSource(new GcsRemoteFileTemplate(new GcsSessionFactory(gcs)));
    adapter.setRemoteDirectory("your-gcs-bucket");
    return adapter;
}
```

Outbound channel adapter

The outbound channel adapter allows files to be written to Google Cloud Storage. When it receives a Message containing a payload of type File, it writes that file to the Google Cloud Storage bucket specified in the adapter.

Here is an example of how to configure a Google Cloud Storage outbound channel adapter.

```
@Bean
@ServiceActivator(inputChannel = "writeFiles")
public MessageHandler outboundChannelAdapter(Storage gcs) {
    GcsMessageHandler outboundChannelAdapter = new GcsMessageHandler(new GcsSessionFactory(gcs));
    outboundChannelAdapter.setRemoteDirectoryExpression(new ValueExpression<>("your-gcs-bucket"));

    return outboundChannelAdapter;
}
```

8. Spring Cloud Sleuth

[Spring Cloud Sleuth](#) is an instrumentation framework for Spring Boot applications. It captures trace informations and can forward traces to services like Zipkin for storage and analysis.

Google Cloud Platform provides its own managed distributed tracing service called [Stackdriver Trace](#). Instead of running and maintaining your own Zipkin instance and storage, you can use Stackdriver Trace to store traces, view trace details, generate latency distributions graphs, and generate performance regression reports.

This Spring Cloud GCP starter can forward Spring Cloud Sleuth traces to Stackdriver Trace without an intermediary Zipkin server.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-trace'
}
```

You must enable Stackdriver Trace API from the Google Cloud Console in order to capture traces. Navigate to the [Stackdriver Trace API](#) for your project and make sure it's enabled.

Note

If you are already using a Zipkin server capturing trace information from multiple platform/frameworks, you also use a [Stackdriver Zipkin proxy](#) to forward those traces to Stackdriver Trace without modifying existing applications.

8.1 Spring Boot Starter for Stackdriver Trace

Spring Boot Starter for Stackdriver Trace uses Spring Cloud Sleuth and auto-configures a [SpanReporter](#) that sends the Sleuth's trace information to Stackdriver Trace.

This starter will send traces asynchronously using a buffered trace consumer that auto-flushes when its buffered trace messages exceed its buffer size or have been buffered for longer than its scheduled delay.

There are several parameters you can use to tune the Stackdriver Trace adapter. All configurations are optional:

Name	Description	Optional	Default value
<code>spring.cloud.gcp.trace.autoconfigure</code>	Auto-configure Spring Cloud Sleuth to send traces to Stackdriver Trace.	Yes	true
<code>spring.cloud.gcp.trace.buffered-consumer.threads</code>	Number of threads to use by the underlying	Yes	4

	gRPC channel to send the trace request to Stackdriver.		
spring.cloud.gcp.trace.size-bytes	Buffer size in bytes. Traces will be flushed to Stackdriver when buffered trace messages exceed this size.	Yes	1% of <code>Runtime.totalMemory()</code>
spring.cloud.gcp.trace.buffered-trace-delay-seconds	Buffered trace messages will be flushed to Stackdriver when buffered longer than scheduled delays (in seconds) even if the buffered message size didn't exceed the buffer size.	Yes	10
spring.cloud.gcp.trace.project-id	Overrides the project ID from the Spring Cloud GCP Module	Yes	
spring.cloud.gcp.trace.credentials.location	Overrides the credentials location from the Spring Cloud GCP Module	Yes	
spring.cloud.gcp.trace.credentials.scopes	Overrides the credentials scopes from the Spring Cloud GCP Module	Yes	

You can use core Spring Cloud Sleuth properties to control Sleuth's sampling rate, etc. Read [Sleuth documentation](#) for more information on Sleuth configurations.

For example, when you are testing to see the traces are going through, you can set the sampling rate to 100%.

```
spring.sleuth.sampler.percentage=1 # Send 100% of the request traces to Stackdriver.
spring.sleuth.web.skipPattern=(^cleanup.*|.+favicon.*) # Ignore some URL paths.
```

Stackdriver Trace requires the use of 128-bit Trace ID. This starter ignores `spring.sleuth.traceId128` property and always uses 128-bit Trace ID.

9. Spring Cloud Config

Spring Cloud GCP makes it possible to use the [Google Runtime Configuration API](#) as a [Spring Cloud Config](#) server to remotely store your application configuration data.

The Spring Cloud GCP Config support is provided via its own Spring Boot starter. It enables the use of the Google Runtime Configuration API as a source for Spring Boot configuration properties.

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.cloud', name: 'spring-cloud-gcp-starter-config'
}
```

9.1 Configuration

The following parameters are configurable in Spring Cloud GCP Config:

Name	Description	Optional	Default value
<code>spring.cloud.gcp.config.enabled</code>	Enables the Config client	Yes	false
<code>spring.cloud.gcp.config.name</code>	Name of your application	No	
<code>spring.cloud.gcp.config.profile</code>	Configuration's Spring profile (e.g., prod)	Yes	default
<code>spring.cloud.gcp.config.timeoutInMillis</code>	Timeout in milliseconds for connecting to the Google Runtime Configuration API	Yes	60000
<code>spring.cloud.gcp.config.projectId</code>	GCP project ID where the Google Runtime Configuration API is hosted	Yes	
<code>spring.cloud.gcp.config.credentials.location</code>	OAuth2 credentials location for authenticating with the Google Runtime Configuration API	Yes	
<code>spring.cloud.gcp.config.credentials.scope</code>	OAuth2 scope for Spring Cloud GCP Config credentials	Yes	https://www.googleapis.com/auth/cloudruntimeconfig

Note

These properties should be specified in a [bootstrap.yml/bootstrap.properties](#) file, rather than the usual `applications.yml/application.properties`.

Note

Also note that core properties as described in [Spring Cloud GCP Core Module](#) do not apply to Spring Cloud GCP Config.

9.2 Quick start

1. Create a configuration in the Google Runtime Configuration API that is called `${spring.cloud.gcp.config.name}_${spring.cloud.gcp.config.profile}`. In other words, if `spring.cloud.gcp.config.name` is `myapp` and `spring.cloud.gcp.config.profile` is `prod`, the configuration should be called `myapp_prod`.

In order to do that, you should have the [Google Cloud SDK](#) installed, own a Google Cloud Project and run the following command:

```
gcloud init # if this is your first Google Cloud SDK run.
gcloud beta runtime-config configs create myapp_prod
gcloud beta runtime-config configs variables set myapp.queue-size 25 --config-name myapp_prod
```

2. Configure your `bootstrap.properties` file with your application's configuration data:

```
spring.cloud.gcp.config.name=myapp
spring.cloud.gcp.config.profile=prod
```

3. Add the `@ConfigurationProperties` annotation to a Spring-managed bean:

```
@Component
@ConfigurationProperties("myapp")
public class SampleConfig {

    private int queueSize;

    public int getQueueSize() {
        return this.queueSize;
    }

    public void setQueueSize(int queueSize) {
        this.queueSize = queueSize;
    }
}
```

When your Spring application starts, the `queueSize` field value will be set to 25 for the above `SampleConfig` bean.

9.3 Refreshing the configuration at runtime

[Spring Boot Actuator](#) enables a `/refresh` endpoint in your application that can be used to refresh the values of all the Spring Cloud Config-managed variables.

To achieve that, add the Spring Boot Actuator dependency.

Maven coordinates:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Gradle coordinates:

```
dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-starter-actuator'
}
```

Add a `@RefreshScope` annotation to your class(es) containing remote configuration properties. Then, if you change the value of the `myapp.queue-size` variable in the `myapp_prod` configuration and hit the `/refresh` endpoint of your application, you can verify that the value of the `queueSize` field has been updated.

Note

If you're developing locally or just not using authentication in your application, you should add `management.security.enabled=false` to your `application.properties` file to allow unrestricted access to the `/refresh` endpoint.