

Spring Cloud GCP Reference Documentation

1.0.0.M3

João André Martins , Jisha Abubaker , Ray Tsang , Mike Eltsufin ,
Artem Bilan , Andreas Berger , Balint Pato , Chengyuan Zhao

Copyright © 2017-2018Pivotal 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	15
Inbound channel adapter	15
Inbound streaming channel adapter	16
Outbound channel adapter	16
8. Spring Cloud Sleuth	17
8.1. Spring Boot Starter for Stackdriver Trace	17
8.2. Integration with Logging	19
9. Stackdriver Logging Support	20
9.1. Logback Support	20
Log via API	21
Log via Console	21
10. Spring Cloud Config	24
10.1. Configuration	24
10.2. Quick start	25
10.3. Refreshing the configuration at runtime	25
11. Spring Data Spanner	27
11.1. Configuration	27

Spanner settings	27
Repository settings	28
Autoconfiguration	28
11.2. Object Mapping	28
Table	28
SpEL expressions for table names	28
Primary Keys	29
Columns	29
Relationships	30
Supported Types	30
Lists	30
Custom types	31
11.3. Spanner Template	32
SQL Query	33
Read	33
Advanced reads	33
Stale read	33
Read from a secondary index	34
Read with offsets and limits	34
Sorting	34
Partial read	34
Summary of options for Query vs Read	34
Write / Update	35
Insert	35
Update	35
Upsert	35
Partial Update	35
Transactions	35
Read/Write Transaction	36
Read-only Transaction	36
11.4. Repositories	37
CRUD Repository	37
Paging and Sorting Repository	38
Query methods by convention	38
Custom SQL query methods	38
Query methods with named queries properties	38
Query methods with annotation	39
REST Repositories	39
12. Cloud Foundry	40

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.M3</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.scope</code>	Auth2 scope for Spring Cloud GCP Pub/Sub 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
------	-------------	----------	---------------

spring.cloud.gcp.storage.create-files	Creates files and buckets on Google Cloud Storage when writes are made to non-existent files	Yes	true
spring.cloud.gcp.storage.credentials.location	OAuth2 credentials for authenticating with the Google Cloud Storage API, if different from the ones in the Spring Cloud GCP Core Module	Yes	
spring.cloud.gcp.storage.credentials.scopes	OAuth2 scope for Spring Cloud GCP Storage 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.instance-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>
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
</dependency>
```

Gradle coordinates:

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

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 `GcpPubSubHeaders.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(GcpPubSubHeaders.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.

To override the default destination you can use the `GcpPubSubHeaders.DESTINATION` header.

```
@Autowired
private MessageChannel pubsubOutputChannel;

public void handleMessage(Message<?> msg) throws MessagingException {
    final Message<?> message = MessageBuilder
        .withPayload(msg.getPayload())
        .setHeader(GcpPubSubHeaders.TOPIC, "customTopic").build();
    pubsubOutputChannel.send(message);
}
```

It is also possible to set an SpEL expression for the topic with the `setTopicExpression()` or `setTopicExpressionString()` methods.

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

Gradle coordinates:

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

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

Spring Cloud GCP Trace does override some Sleuth configurations:

- Always uses 128-bit Trace IDs. This is required by Stackdriver Trace.
- Does not use Span joins. Span joins will share the span ID between the client and server Spans. Stackdriver requires that every Span ID within a Trace to be unique, so Span joins are not supported.
- Uses `StackdriverHttpClientParser` and `StackdriverHttpServerParser` by default to populate Stackdriver related fields.

8.2 Integration with Logging

Logs can also be associated with traces. See [Stackdriver Logging Support](#) for how to configure logging such that it carries Trace ID metadata.

9. Stackdriver Logging Support

Maven coordinates, using Spring Cloud GCP BOM:

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

Gradle coordinates:

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

[Stackdriver Logging](#) is the managed logging service provided by Google Cloud Platform.

This module provides support for associating a web request trace ID with the corresponding log entries. This allows grouping of log messages by request.

Note

Due to the way logging is set up, the GCP project ID and credentials defined in `application.properties` are ignored. Instead, you should set the `GOOGLE_CLOUD_PROJECT` and `GOOGLE_APPLICATION_CREDENTIALS` environment variables to the project ID and credentials private key location, respectively.

`TraceIdLoggingWebMvcInterceptor` extracts the request trace ID from an HTTP request using a `TraceIdExtractor` and stores it in a thread-local of the [TraceLoggingEnhancer](#) which can then be used in a logging appender to add the trace ID metadata to log messages.

There are implementations provided for `TraceIdExtractor`:

Extractor	Description
<code>XCloudTraceIdExtractor</code>	Checks the X-Cloud-Trace-Context HTTP header that is automatically added to HTTP requests sent to applications hosted on GCP services such as App Engine and GKE
<code>ZipkinTraceIdExtractor</code>	Checks the X-B3-TraceId header
<code>CompositeTraceIdExtractor</code>	Instantiated with a list of other <code>TraceIdExtractor</code> , and provides the first trace ID found using these extractors in the given order

`LoggingWebMvcConfigurer` configuration class is also provided to help register the `TraceIdLoggingWebMvcInterceptor` in Spring MVC applications.

Currently, [Java Util Logging \(JUL\)](#) and [Logback](#) are supported.

9.1 Logback Support

There are 2 possibilities to log to Stackdriver via this library with Logback:

Log via API

For Logback, a `org/springframework/cloud/gcp/autoconfigure/logging/logback-appender.xml` file is made available for import to make it easier to configure the Logback appender.

Your configuration may then look something like this:

```
<configuration>
  <include resource="org/springframework/cloud/gcp/autoconfigure/logging/logback-appender.xml" />

  <root level="INFO">
    <appender-ref ref="STACKDRIVER" />
  </root>
</configuration>
```

`STACKDRIVER_LOG_NAME` and `STACKDRIVER_LOG_FLUSH_LEVEL` environment variables can be used to customize the STACKDRIVER appender.

Also see the [spring-cloud-gcp-starter-logging](#) module.

Log via Console

If you run your Spring Boot application in a Kubernetes cluster in the Google Cloud (GKE) or on App Engine flexible environment, you can log JSON directly from the console by including `org/springframework/cloud/gcp/logging/logback-json-appender.xml`. The `traceId` will be set correctly.

You need the additional dependency:

```
<dependency>
  <groupId>ch.qos.logback.contrib</groupId>
  <artifactId>logback-json-classic</artifactId>
  <version>0.1.5</version>
</dependency>
```

Your Logback configuration may then look something like this:

```
<configuration>
  <property name="projectId" value="test-project"/>
  <include resource="org/springframework/cloud/gcp/logging/logback-json-appender.xml" />

  <root level="INFO">
    <appender-ref ref="CONSOLE_JSON"/>
  </root>
</configuration>
```

If you want to have more control over the log output, you can also configure the `ConsoleAppender` yourself. The following properties are available:

Property	Default Value	Description
<code>projectId</code>		Only required for logging the correct <code>traceId</code> : <code>projects/[PROJECT-ID]/traces/[TRACE-ID]</code> . If <code>projectId</code> is not set, the logged <code>traceId</code> will match the one provided by <code>spring-cloud-gcp-trace</code>

Property	Default Value	Description
includeTraceId	true	Should the <code>traceId</code> be included
includeSpanId	true	Should the <code>spanId</code> be included
includeLevel	true	Should the severity be included
includeThreadName	true	Should the thread name be included
includeMDC	true	Should all MDC properties be included. The MDC properties <code>X-B3-TraceId</code> , <code>X-B3-SpanId</code> and <code>X-Span-Export</code> provided by Spring Sleuth will get excluded as they get handled separately
includeLoggerName	true	Should the name of the logger be included
includeFormattedMessage	true	Should the formatted log message be included.
includeExceptionInMessage	true	Should the stacktrace be appended to the formatted log message. This setting is only evaluated if <code>includeFormattedMessage</code> is true
includeContextName	true	Should the logging context be included
includeMessage	false	Should the log message with blank placeholders be included
includeException	false	Should the stacktrace be included as a own field

This is an example of such an Logback configuration:

```
<configuration >
  <property name="projectId" value="${projectId:-${GOOGLE_CLOUD_PROJECT}}"/>

  <appender name="CONSOLE_JSON" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
      <layout class="org.springframework.cloud.gcp.logging.StackdriverJsonLayout">
        <projectId>${projectId}</projectId>

        <!--<includeTraceId>true</includeTraceId>-->
        <!--<includeSpanId>true</includeSpanId>-->
        <!--<includeLevel>true</includeLevel>-->
        <!--<includeThreadName>true</includeThreadName>-->
        <!--<includeMDC>true</includeMDC>-->
        <!--<includeLoggerName>true</includeLoggerName>-->
        <!--<includeFormattedMessage>true</includeFormattedMessage>-->
```

```
        <!--<includeExceptionInMessage>true</includeExceptionInMessage>-->
        <!--<includeContextName>true</includeContextName>-->
        <!--<includeMessage>false</includeMessage>-->
        <!--<includeException>false</includeException>-->
    </layout>
</encoder>
</appender>
</configuration>
```

10. 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'
}
```

10.1 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.

10.2 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.

10.3 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.

11. Spring Data Spanner

[Spring Data](#) is an abstraction for storing and retrieving POJOs in numerous storage technologies. Spring Cloud GCP adds Spring Data support for [Google Cloud Spanner](#).

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

Maven coordinates for the Spring Boot Starter for Spanner, using Spring Cloud GCP BOM:

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

Gradle coordinates:

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

This setup takes care of bringing in the latest compatible version of Cloud Java Spanner libraries as well.

11.1 Configuration

To setup Spring Data Spanner, you have to configure the following:

- Setup the connection details to Google Cloud Spanner.
- Enable Spring Data Repositories (optional).

Spanner settings

You can use [Spring Boot Starter for Spring Data Spanner](#) to autoconfigure Google Cloud Spanner in your Spring application. It contains all the necessary setup that makes it easy to authenticate with your Google Cloud project. The following configuration options are available:

Name	Description	Optional	Default value
spring.cloud.gcp.spanner.instance.id	Spanner instance to use	No	
spring.cloud.gcp.spanner.database	Spanner database to use	No	
spring.cloud.gcp.spanner.project.id	GCP project ID where the Google Cloud Spanner API is hosted, if different from the one in the Spring Cloud GCP Core Module	Yes	
spring.cloud.gcp.spanner.credentials.location	OAuth2 credentials for authenticating with the Google Cloud Spanner API, if different from the ones in the Spring	Yes	

	Cloud GCP Core Module		
spring.cloud.gcp.spanner.credentials.scope	@Auth2 scope for Spring Cloud GCP Spanner credentials	Yes	https://www.googleapis.com/auth/spanner.data

Repository settings

Spring Data Repositories can be configured via the `@EnableSpannerRepositories` annotation on your main `@Configuration` class. With our Spring Boot Starter for Spring Data Spanner, `@EnableSpannerRepositories` is automatically added. It is not required to add it to any other class, unless there is a need to override finer grain configuration parameters provided by [@EnableSpannerRepositories](#).

Autoconfiguration

Our Spring Boot autoconfiguration creates the following beans available in the Spring application context:

- an instance of `SpannerTemplate`
- an instance of all user defined repositories extending `CrudRepository` or `PagingAndSortingRepository`, when repositories are enabled
- an instance of `DatabaseClient` from the Google Cloud Java Client for Spanner, for convenience and lower level API access

11.2 Object Mapping

Spring Data Spanner allows you to map domain POJOs to Spanner tables via annotations:

```
@Table(name = "traders")
public class Trader {

    @PrimaryKey
    @Column(name = "trader_id")
    String traderId;

    String firstName;

    String lastName;
}
```

Table

The `@Table` annotation can provide the name of the Spanner table that stores instances of the annotated class, one per row. This annotation is optional, and if not given, the name of the table is inferred from the class name with the first character uncapitalized.

SpEL expressions for table names

In some cases, you might want the `@Table` table name to be determined dynamically. To do that, you can use [Spring Expression Language](#).

For example:

```
@Table(name = "trades_#{tableNameSuffix}")
public class Trade {
    // ...
}
```

The table name will be resolved only if the `tableNameSuffix` value/bean in the Spring application context is defined. For example, if `tableNameSuffix` has the value "123", the table name will resolve to `trades_123`.

Primary Keys

For a simple table, you may only have a primary key consisting of a single column. Even in that case, the `@PrimaryKey` annotation is required. `@PrimaryKey` identifies the one or more ID properties corresponding to the primary key.

Spanner has first class support for composite primary keys of multiple columns. You have to annotate all of your POJO's fields that the primary key consists of with `@PrimaryKey` as below:

```
@Table(name = "trades")
public class Trade {
    @PrimaryKey(keyOrder = 2)
    @Column(name = "trade_id")
    private String tradeId;

    @PrimaryKey(keyOrder = 1)
    @Column(name = "trader_id")
    private String traderId;

    private String action;

    private Double price;

    private Double shares;

    private String symbol;
}
```

The `keyOrder` parameter of `@PrimaryKey` identifies the properties corresponding to the primary key columns in order, starting with 1 and increasing consecutively. Order is important and must reflect the order defined in the Spanner schema. In our example the DDL to create the table and its primary key is as follows:

```
CREATE TABLE trades (
    trader_id STRING(MAX),
    trade_id STRING(MAX),
    action STRING(15),
    symbol STRING(10),
    price FLOAT64,
    shares FLOAT64
) PRIMARY KEY (trader_id, trade_id)
```

Spanner does not have automatic ID generation. For most use-cases, sequential IDs should be used with caution to avoid creating data hotspots in the system. Read [Spanner Primary Keys documentation](#) for a better understanding of primary keys and recommended practices.

Columns

All accessible properties on POJOs are automatically recognized as a Spanner column. Column naming is generated by the `PropertyNameFieldNameNamingStrategy` by default defined on the `SpannerMappingContext` bean. The `@Column` annotation optionally provides a different column name than that of the property.

Relationships

Currently there is no support to map relationships between objects. I.e., currently we do not have ways to establish parent-children relationships directly via annotations. This feature is actively being worked on.

Supported Types

Spring Data Spanner supports the following types for regular fields:

- `com.google.cloud.ByteArray`
- `com.google.cloud.Date`
- `com.google.cloud.Timestamp`
- `java.lang.Boolean`, `boolean`
- `java.lang.Double`, `double`
- `java.lang.Long`, `long`
- `java.lang.Integer`, `int`
- `java.lang.String`
- `double[]`
- `long[]`
- `boolean[]`
- `java.util.Date`
- `java.util.Instant`
- `java.sql.Date`

Lists

Spanner supports `ARRAY` types for columns. `ARRAY` columns are mapped to `List` fields in POJOS. When using `List`, the `@ColumnInnerType` annotation is required to specify what is the type of the elements.

Example:

```
@ColumnInnerType(innerType = Double.class)
List<Double> curve;
```

Spring Data Spanner supports the following inner types:

- `com.google.cloud.ByteArray`
- `com.google.cloud.Date`
- `com.google.cloud.Timestamp`
- `java.lang.Boolean`, `boolean`
- `java.lang.Double`, `double`
- `java.lang.Long`, `long`

- `java.lang.Integer`, `int`
- `java.lang.String`
- `java.util.Date`
- `java.util.Instant`
- `java.sql.Date`

Custom types

Custom converters can be used extending the type support for user defined types.

1. Converters need to implement the `org.springframework.core.convert.converter.Converter` interface both directions.
2. The user defined type needs to be mapped to one the basic types supported by Spanner:
 - `com.google.cloud.ByteArray`
 - `com.google.cloud.Date`
 - `com.google.cloud.Timestamp`
 - `java.lang.Boolean`, `boolean`
 - `java.lang.Double`, `double`
 - `java.lang.Long`, `long`
 - `java.lang.String`
 - `double[]`
 - `long[]`
 - `boolean[]`
3. An instance of both Converters needs to be passed to a `MappingSpannerConverter`, which then has to be made available as a `@Bean` for `SpannerConverter`.

For example:

We would like to have a field of type `Person` on our `Trade` POJO:

```
@Table(name = "trades")
public class Trade {
    //...
    Person person;
    //...
}
```

Where `Person` is a simple class:

```
public class Person {

    public String firstName;
    public String lastName;

}
```

We have to define the two converters:

```
public class PersonWriteConverter implements Converter<Person, String> {

    @Override
    public String convert(Person person) {
        return person.firstName + " " + person.lastName;
    }
}

public class PersonReadConverter implements Converter<String, Person> {

    @Override
    public Person convert(String s) {
        Person person = new Person();
        person.firstName = s.split(" ")[0];
        person.lastName = s.split(" ")[1];
        return person;
    }
}
```

That will be configured in our @Configuration file:

```
@Configuration
public class ConverterConfiguration {

    @Bean
    public SpannerConverter spannerConverter(SpannerMappingContext spannerMappingContext) {
        return new MappingSpannerConverter(spannerMappingContext,
            Arrays.asList(new PersonWriteConverter()),
            Arrays.asList(new PersonReadConverter()));
    }
}
```

11.3 Spanner Template

SpannerOperations and its implementation, SpannerTemplate, provides the Template pattern familiar to Spring developers. It provides:

- Resource management
- One-stop-shop to Spanner operations with the Spring Data POJO mapping and conversion features
- Exception conversion

Using the autoconfigure provided by our Spring Boot Starter for Spanner, your Spring application context will contain a fully configured SpannerTemplate object that you can easily autowire in your application:

```
@SpringBootApplication
public class SpannerTemplateExample {

    @Autowired
    SpannerTemplate spannerTemplate;

    public void doSomething() {
        this.spannerTemplate.delete(Trade.class, KeySet.all());
        //...
        Trade t = new Trade();
        //...
        this.spannerTemplate.insert(t);
        //...
        List<Trade> tradesByAction = spannerTemplate.findAll(Trade.class);
        //...
    }
}
```

```
}
```

The Template API provides convenience methods for:

- [Reads](#), and by providing `SpannerReadOptions` and `SpannerQueryOptions`
 - Stale read
 - Read with secondary indices
 - Read with limits and offsets
 - Read with sorting
- [Queries](#)
- DML operations (delete, insert, update, upsert)
- Partial reads
 - You can define a set of columns to be read into your entity
- Partial writes
 - Persist only a few properties from your entity
- Read-only transactions
- Locking read-write transactions

SQL Query

Spanner has SQL support for running read-only queries. All the query related methods start with `query` on `SpannerTemplate`. Using `SpannerTemplate` you can execute SQL queries that map to POJOs:

```
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT * FROM trades"));
```

Read

Spanner exposes a [Read API](#) for reading single row or multiple rows in a table or in a secondary index.

Using `SpannerTemplate` you can execute reads, for example:

```
List<Trade> trades = this.spannerTemplate.readAll(Trade.class);
```

Main benefit of reads over queries is reading multiple rows of a certain pattern of keys is much easier using the features of the [KeySet](#) class.

Advanced reads

Stale read

All reads and queries are **strong reads** by default. A **strong read** is a read at a current timestamp and is guaranteed to see all data that has been committed up until the start of this read. A **stale read** on the other hand is read at a timestamp in the past. Cloud Spanner allows you to determine how current the data should be when you read data. With `SpannerTemplate` you can specify the `Timestamp`

by setting it on `SpannerQueryOptions` or `SpannerReadOptions` to the appropriate read or query methods:

Reads:

```
// a read with options:
SpannerReadOptions spannerReadOptions = new SpannerReadOptions().setTimestamp(Timestamp.now());
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Queries:

```
// a query with options:
SpannerQueryOptions spannerQueryOptions = new SpannerQueryOptions().setTimestamp(Timestamp.now());
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT * FROM trades"),
spannerQueryOptions);
```

Read from a secondary index

Using a [secondary index](#) is available for Reads via the Template API and it is also implicitly available via SQL for Queries.

The following shows how to read rows from a table using a [secondary index](#) simply by setting index on `SpannerReadOptions`:

```
SpannerReadOptions spannerReadOptions = new SpannerReadOptions().setIndex("TradesByTrader");
List<Trade> trades = this.spannerTemplate.readAll(Trade.class, spannerReadOptions);
```

Read with offsets and limits

Limits and offsets are only supported by Queries. The following will get only the first two rows of the query:

```
SpannerQueryOptions spannerQueryOptions = new SpannerQueryOptions().setLimit(2).setOffset(3);
List<Trade> trades = this.spannerTemplate.query(Trade.class, Statement.of("SELECT * FROM trades"),
spannerQueryOptions);
```

Note that the above is equivalent of executing `SELECT * FROM trades LIMIT 2 OFFSET 3`.

Sorting

Reads don't support sorting. Queries on the Template API support sorting through standard SQL and also via Spring Data Sort API:

```
List<Trade> trades = this.spannerTemplate.queryAll(Trade.class, Sort.by("action"));
```

Partial read

Partial read is only possible when using Queries. In case the rows returned by query have fewer columns than the entity that it will be mapped to, Spring Data will map the returned columns and leave the rest as they of the columns are.

Summary of options for Query vs Read

Feature	Query supports it	Read supports it
SQL	yes	no
Partial read	yes	no
Limits	yes	no

Offsets	yes	no
Secondary index	yes	yes
Read using index range	no	yes
Sorting	yes	no

Write / Update

The write methods of `SpannerOperations` accept a POJO and writes all of its properties to Spanner. The corresponding Spanner table and entity metadata is obtained from the given object's actual type.

If a POJO was retrieved from Spanner and its primary key properties values were changed and then written or updated, the operation will occur as if against a row with the new primary key values. The row with the original primary key values will not be affected.

Insert

The `insert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if a row with the POJO's primary key already exists in the table.

```
Trade t = new Trade();
this.spannerOperations.insert(t);
```

Update

The `update` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner, which means the operation will fail if the POJO's primary key does not already exist in the table.

```
// t was retrieved from a previous operation
this.spannerOperations.update(t);
```

Upsert

The `upsert` method of `SpannerOperations` accepts a POJO and writes all of its properties to Spanner using update-or-insert.

```
// t was retrieved from a previous operation or it's new
this.spannerOperations.upsert(t);
```

Partial Update

The update methods of `SpannerOperations` operate by default on all properties within the given object, but also accept `String[]` and `Optional<Set<String>>` of column names. If the `Optional` of set of column names is empty, then all columns are written to Spanner. However, if the `Optional` is occupied by an empty set, then no columns will be written.

```
// t was retrieved from a previous operation or it's new
this.spannerOperations.update(t, "symbol", "action");
```

Transactions

`SpannerOperations` provides methods to run `java.util.Function` objects within a single transaction while making available the read and write methods from `SpannerOperations`.

Read/Write Transaction

Read and write transactions are provided by `SpannerOperations` via the `performReadWriteTransaction` method:

```
@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
    return mySpannerOperations.performReadWriteTransaction(
        transActionSpannerOperations -> {
            // work with transActionSpannerOperations here. It is also a SpannerOperations object.

            return "transaction completed";
        }
    );
}
```

The `performReadWriteTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads, because all reads and writes happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`.

As these read-write transactions are locking, it is recommended that you use the `performReadOnlyTransaction` if your function does not perform any writes.

Read-only Transaction

The `performReadOnlyTransaction` method is used to perform read-only transactions using a `SpannerOperations`:

```
@Autowired
SpannerOperations mySpannerOperations;

public String doWorkInsideTransaction() {
    return mySpannerOperations.performReadOnlyTransaction(
        transActionSpannerOperations -> {
            // work with transActionSpannerOperations here. It is also a SpannerOperations object.

            return "transaction completed";
        }
    );
}
```

The `performReadOnlyTransaction` method accepts a `Function` that is provided an instance of a `SpannerOperations` object. This method also accepts a `ReadOptions` object, but the only attribute used is the timestamp used to determine the snapshot in time to perform the reads in the transaction. If the timestamp is not set in the read options the transaction is run against the current state of the database. The final returned value and type of the function is determined by the user. You can use this object just as you would a regular `SpannerOperations` with a few exceptions:

- Its read functionality cannot perform stale reads, because all reads happen at the single point in time of the transaction.
- It cannot perform sub-transactions via `performReadWriteTransaction` or `performReadOnlyTransaction`.

- It cannot perform any write operations.

Because read-only transactions are non-locking and can be performed on points in time in the past, these are recommended for functions that do not perform write operations.

11.4 Repositories

[Spring Data Repositories](#) are a powerful abstraction that can save you a lot of boilerplate code.

For example:

```
public interface TraderRepository extends CrudRepository<Trader, String> {
}
```

Spring Data generates a working implementation of the specified interface, which can be conveniently autowired into an application.

The `Trader` type parameter to `CrudRepository` refers to the underlying domain type. The second type parameter, `String` in this case, refers to the type of the key of the domain type.

For POJOs with a composite primary key, this ID type parameter can be any descendant of `Object[]` compatible with all primary key properties, any descendant of `Iterable`, or `com.google.cloud.spanner.Key`. If the domain POJO type only has a single primary key column, then the primary key property type can be used or the `Key` type.

For example in case of Trades, that belong to a Trader, `TradeRepository` would look like this:

```
public interface TradeRepository extends CrudRepository<Trade, String[]> {
}
```

```
public class MyApplication {

    @Autowired
    SpannerOperations spannerOperations;

    @Autowired
    StudentRepository studentRepository;

    public void demo() {

        this.tradeRepository.deleteAll(); //defined on CrudRepository
        String traderId = "demo_trader";
        Trade t = new Trade();
        t.symbol = stock;
        t.action = action;
        t.traderId = traderId;
        t.price = 100.0;
        t.shares = 12345.6;
        this.spannerOperations.insert(t); //defined on CrudRepository

        Iterable<Trade> allTrades = this.tradeRepository.findAll(); //defined on CrudRepository

        int count = this.tradeRepository.countByAction("BUY");

    }
}
```

CRUD Repository

`CrudRepository` methods work as expected, with one thing Spanner specific: the `save` and `saveAll` methods work as update-or-insert.

Paging and Sorting Repository

You can also use `PagingAndSortingRepository` with `Spanner` Spring Data. The sorting and pageable `findAll` methods available from this interface operate on the current state of the `Spanner` database. As a result, beware that the state of the database (and the results) might change when moving page to page.

Query methods by convention

```
public interface TradeRepository extends CrudRepository<Trade, String[]> {
    List<Trade> findByAction(String action);

    int countByAction(String action);

    // Named methods are powerful, but can get unwieldy
    List<Trade> findTop3DistinctByActionAndSymbolOrTraderIdOrderBySymbolDesc(
        String action, String symbol, String traderId);
}
```

In the example above, the [query methods](#) in `TradeRepository` are generated based on the name of the methods, using the [Spring Data Query creation naming convention](#).

`List<Trade> findByAction(String action)` would translate to a `SELECT * FROM trades WHERE action = ?`.

The `function` `List<Trade> findTop3DistinctByActionAndSymbolOrTraderIdOrderBySymbolDesc(String action, String symbol, String traderId);` will be translated as the equivalent of this SQL query:

```
SELECT DISTINCT * FROM trades
WHERE ACTION = ? AND SYMBOL = ? AND TRADER_ID = ?
ORDER BY SYMBOL DESC
LIMIT 3
```

Custom SQL query methods

The example above for `List<Trade> fetchByActionNamedQuery(String action)` does not match the [Spring Data Query creation naming convention](#), so we have to map a parametrized `Spanner` SQL query to it.

The SQL query for the method can be mapped to repository methods in one of two ways:

- `namedQueries` properties file
- using the `@Query` annotation

Query methods with named queries properties

By default, the `namedQueriesLocation` attribute on `@EnableSpannerRepositories` points to the `META-INF/spanner-named-queries.properties` file. You can specify the query for a method in the properties file by providing the SQL as the value for the "interface.method" property:

```
Trade.fetchByActionNamedQuery=SELECT * FROM trades WHERE trades.action = @tag0
```

```
public interface TradeRepository extends CrudRepository<Trade, String[]> {
    // This method uses the query from the properties file instead of one generated based on name.
    List<Trade> fetchByActionNamedQuery(String action);
}
```

Query methods with annotation

Using the `@Query` annotation:

```
public interface TradeRepository extends CrudRepository<Trade, String[]> {
    @Query("SELECT * FROM trades WHERE trades.action = @tag0")
    List<Trade> fetchByActionNamedQuery(String action);
}
```

Table names can be used directly. For example, "trades" in the above example. Alternatively, table names can be resolved from the `@Table` annotation on domain classes as well. In this case, the query should refer to table names with fully qualified class names between `:` characters: `:fully.qualified.ClassName:.` A full example would look like:

```
@Query("SELECT * FROM :com.example.Trade: WHERE trades.action = @tag0")
List<Trade> fetchByActionNamedQuery(String action);
```

This allows table names evaluated with SpEL to be used in custom queries.

REST Repositories

When running with Spring Boot, repositories can act as REST services by simply annotating them:

```
@RepositoryRestResource(collectionResourceRel = "trades", path = "trades")
public interface TradeRepository extends CrudRepository<Trade, String[]> {
}
```

The `@RepositoryRestResource` annotation makes this repository available via REST. For example, you can retrieve all Trade objects in the repository by using `curl http://<server>:<port>/trades`, or any specific trade via `curl http://<server>:<port>/trades/<trader_id>,<trade_id>`.

The separator between your primary key components, `id` and `trader_id` in this case, is a comma by default, but can be configured to any string not found in your key values by extending the `SpannerKeyIdConverter` class:

```
@Component
class MySpecialIdConverter extends SpannerKeyIdConverter {

    @Override
    protected String getUrlIdSeparator() {
        return ",";
    }
}
```

You can also write trades using `curl -XPOST -H"Content-Type: application/json" -d@test.json http://<server>:<port>/trades/` where the file `test.json` holds the JSON representation of a Trade object.

Include this dependency in your `pom.xml` to enable Spring Data REST Repositories:

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

12. Cloud Foundry

Spring Cloud GCP provides support for Cloud Foundry's [GCP Service Broker](#). Our Pub/Sub, Spanner, Storage and Stackdriver Trace starters are Cloud Foundry aware and retrieve properties like project ID, credentials, etc., that are used in auto configuration.

In cases like Pub/Sub's topic and subscription, or Storage's bucket name, where those parameters are not used in auto configuration, you can fetch them using the VCAP mapping provided by Spring Boot. For example, to retrieve the provisioned Pub/Sub topic, you can use the `vcap.services.mypubsub.credentials.topic_name` property from the application environment.

Note

If the same service is bound to the same application more than once, the auto configuration will not be able to choose among bindings and will not be activated for that service.