FTG FEARLESS TECHNOLOGY GROUP

commercetools
Next generation commerce

Building commercetools
customizations using Subscriptions and the Google Cloud Platform

www.commercetools.com

Table of Contents

Introduction

> ***"By using Event Messages you can easily decouple senders and receivers both in terms of identity (you broadcast events without caring who responds to them) and time (events can be queued and forwarded when the receiver is ready to process them). Such architectures offer a great deal for scalability and modifiability due to this loose coupling."***
>
> – Focusing on Events, Martin Fowler

commercetools is a dynamically extensible, cloud-native commerce solution. It allows retailers to sculpt a solution that fits their unique needs today, and is flexible to support their evolving business strategy tomorrow.

There are many powerful extensibility features built into commercetools that handle a wide variety of use cases. For an overview of them, see Building commercetools customizations - Overview.

In this whitepaper we will do a deep dive on one powerful technique for customizing commercetools: Subscriptions.

What are Subscriptions?

Subscriptions allow you to trigger custom asynchronous background processing in response to an event on the commercetools platform.

> ***"Subscriptions allow you to be notified of new messages or changes via a Message Queue of your choice."***
>
> – Subscriptions, Platform Documentation, commercetools

Because Subscriptions execute asynchronously based on events emitted from the platform, they allow your custom solutions to be loosely coupled to commercetools. This greatly reduces the risk of your code impacting commercetools API execution and performance.

commercetools differentiates between **messages** and **changes**. A single subscription can listen to both depending on the resource. Changes are straight-forward: events are fired whenever the subscribed resource type is created, updated or deleted. Messages are more specific:

> *"A message represents a change or an action performed on a resource (like an Order or a Product). Messages can be seen as a subset of the change history for a resource inside a project. It is a subset because not all changes on resources result in messages. Messages can be pulled via a REST API, or they can be pushed into a Message Queue by defining a Subscription."*
>
> – Message Types, Platform Documentation, commercetools

There are many use cases where the asynchronous processing of an event is the right approach; for instance, sending order status emails in response to order events. In this use case, commercetools will fire events you can listen for as the `orderState` moves from Open to Confirmed and finally to either Complete or Cancelled.

To take advantage of this technique, you must configure your subscription via commercetools' `/{projectKey}/subscriptions` endpoint. You specify the **destination** for the subscription which, as of this writing, can be any of these message queues: AWS SQS, AWS SNS, Azure Service Bus, Azure Event Grid or Google Cloud Pub/Sub. You also define the array of **messages** and **changes** you want to subscribe to.

Bottom line, Subscriptions provide a lot of power! If you're in doubt whether Subscriptions are the best approach for you, contact FTG and we can help you find the best path.

Let's drill down on a sample use case to see how to exploit Subscriptions.

Implementation Example

Our example implementation will focus on commercetools orders and will use Google's Pub/Sub, Cloud Function and Cloud Firestore services.

Use Case

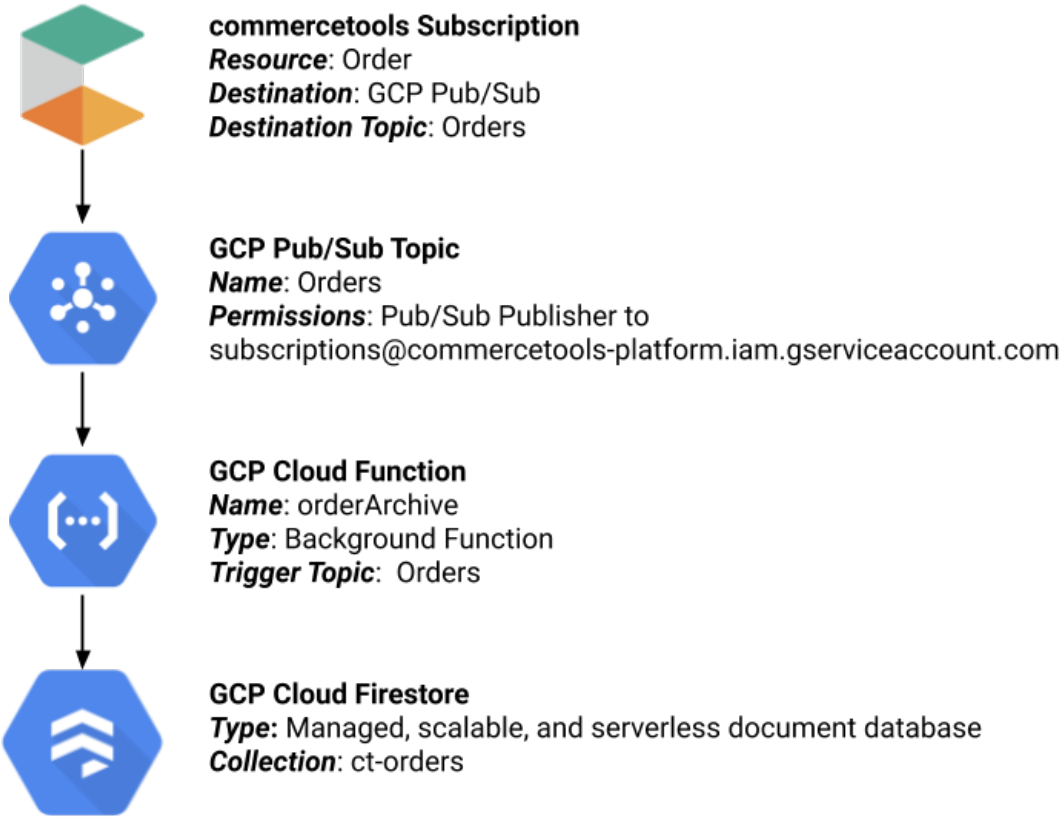
A retailer wishes to archive all commercetools order updates on their own Google Cloud Platform (GCP) instance so they can independently use, query and analyze the data in their existing environment.

Problem

The retailer has not integrated their GCP instance with commercetools.

Solution

Create a commercetools Subscription that will send Order updates to a GCP Pub/Sub Topic where they will be saved to a Cloud Firestore collection by a Cloud Function.



Implementation Steps

Our example implementation is thorough! We will show you how to:

A. Set up a GCP Project,
B. Set up GCP Cloud Firestore,
C. Create a GCP Cloud Function,
D. Configure and Deploy the Cloud Function,
E. Set up the GCP Pub/Sub Topic,
F. Create a commercetools project,
G. Load commercetools sample data,
H. Set up commercetools Subscription
I. Test commercetools/GCP Integration,
J. Provide you with resources if you need help.

Step A: Set up a GCP Project.

If you choose to follow the steps in this example on your own, you will need a GCP account, a GCP project and the Google Cloud SDK. If you're missing any of these things, no worries, you can start running all three for free.

Set up a Google Cloud Account

Go to https://cloud.google.com/free/ to check out the wide array of services available to you on GCP. New customers get a large credit applied to their account to allow for plenty of experimentation before needing to spend money. Click on the "Get Started for Free" link to sign in and get your account up and running.

Set up a Google Cloud Project

Once you have an account, you can set up a project for our example implementation. If you just created a new GCP account, you likely have a default project you can go ahead and use. If you'd like to create a new project, go to https://console.cloud.google.com/projectcreate and set one up by entering a Project name, Organization and Location. In either case, your project will have a "Project ID" which we will use later on so note it for future reference. You can also find your Project ID on your dashboard at https://console.cloud.google.com/home/dashboard.

Install the Google Cloud SDK

The Cloud SDK gives you tools and libraries for interacting with Google Cloud products and services. You can follow Google's installation instructions for your operating system by going to https://cloud.google.com/sdk/docs/install.

Authenticate using Google Cloud SDK

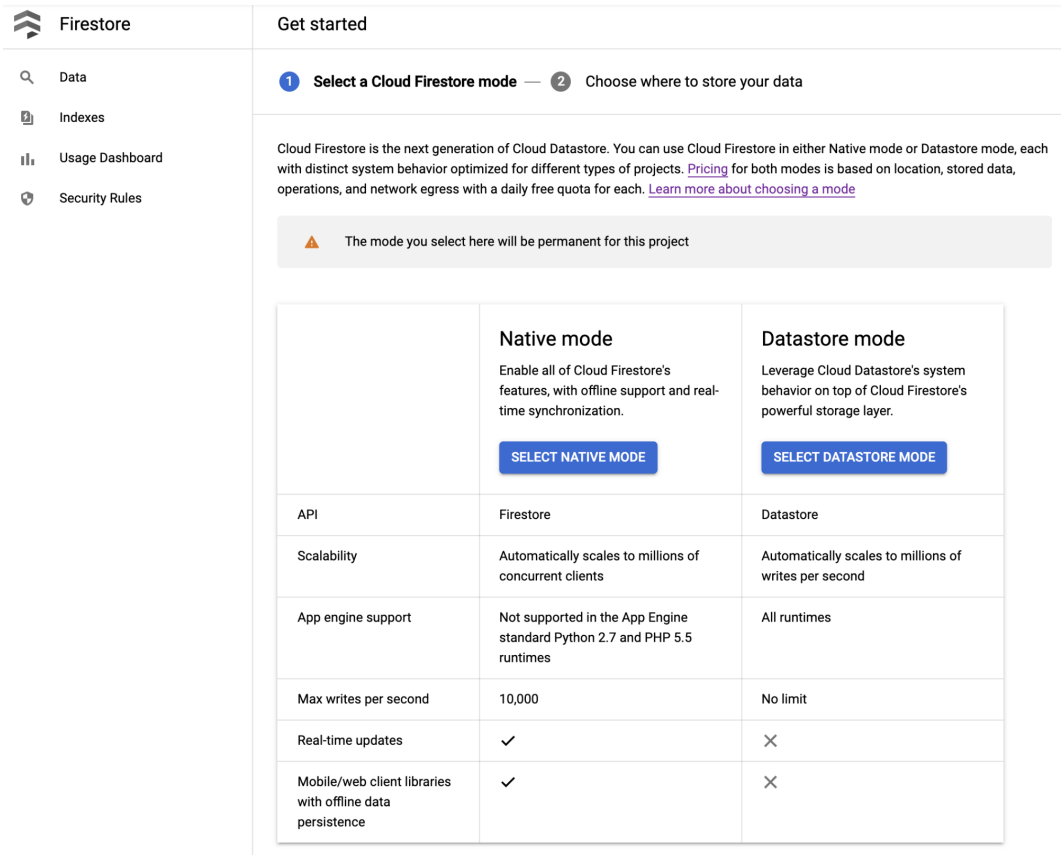
You need to authorize the gcloud command line interface and the SDK before you can use them. If you've not already done so, you can run `gcloud init` to authorize; see Authorizing Cloud SDK tools for details.

Step B: Set up GCP Cloud Firestore.

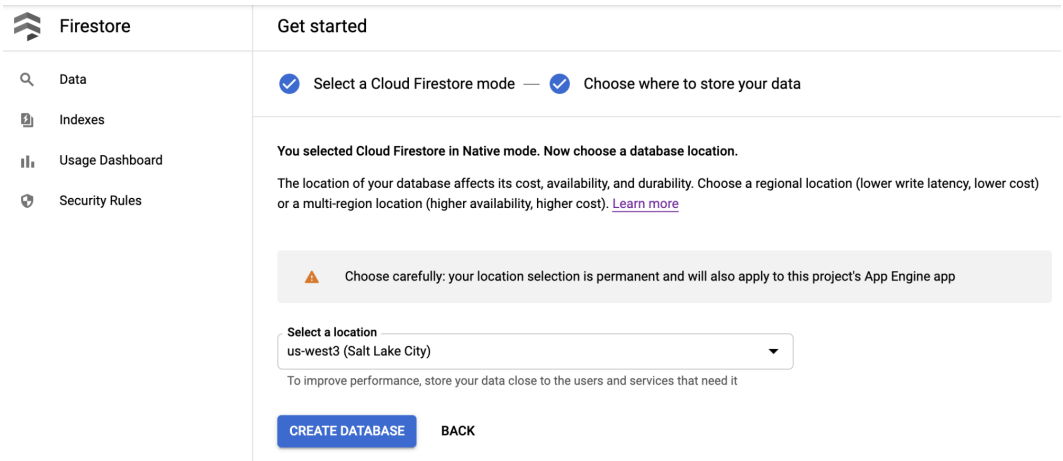
Now that we have a GCP project up and running, let's set up a database where we can persist our Subscription content. Google provides a wide number of database options. We will use Cloud Firestore in this example.

> ***"Firestore is a NoSQL document database built for automatic scaling, high performance, and ease of application development. While the Firestore interface has many of the same features as traditional databases, as a NoSQL database it differs from them in the way it describes relationships between data objects."***
>
> – Firestore documentation

The Subscription messages we will consume from commercetools fit naturally into a document database. This makes Cloud Firestore a good fit for us. Go to https://console.cloud.google.com/firestore/ where you should see a "Get started" screen:



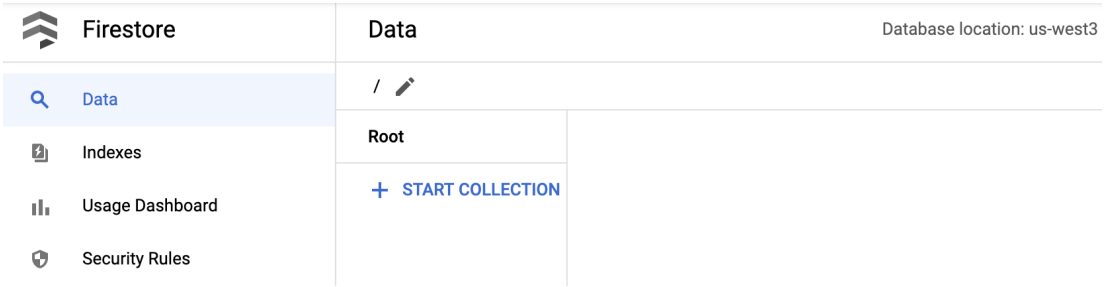
We will select Native Mode but you can learn more about both options provided here by visiting Choosing between Native mode and Datastore mode. The next screen will prompt you for a location.



This example uses “us-west3” but any location will work. If you plan to use this instance heavily then you should spend time reviewing Google’s documentation on Locations and Pricing by Location to be sure you pick the right location for you.

> *“This location setting is your project’s default Google Cloud Platform (GCP) resource location. Note that this location will be used for GCP services in your project that require a location setting, specifically, your default Cloud Storage bucket and your App Engine app (which is required if you use Cloud Scheduler). Warning:* **After you set your project’s default GCP resource location, you cannot change it.”**
>
> – Firestore Quickstart documentation

After clicking **Create Database** you will get a message like, “Creating your database! Initializing Cloud Firestore in Native mode services in us-west3 - this usually takes a few minutes. You’ll be redirected to your database once it’s ready.” Once initialization completes, the console provides an interface where you can perform CRUD operations on Firestore collections. Feel free to experiment here. We will programmatically create a collection and write documents to it in upcoming steps.



Step C: Create a GCP Cloud Function.

Let’s look at some code. Open a command line (our examples use bash) and issue these three commands to clone and initialize FTG’s commercetools-gcp-subscribe repository:

```
$ git clone https://github.com/FearlessTechnologyGroup/commercetools-
gcp-subscribe
$ cd commercetools-gcp-subscribe/
$ npm install
```

This repo contains a cloud function that accepts a commercetools Message and persists it to our Cloud Firestore database. The function is implemented in index.js and has two dependencies, @google-cloud/firestore and joi:

```
// The Node.js Server SDK for Google Cloud Firestore:
const Firestore = require('@google-cloud/firestore');

// Schema description language and data validator:
const Joi = require('joi');
```

We use two environment variables:

```
const PROJECTID = process.env.PROJECTID; // your GCP project name
const COLLECTION_NAME = process.env.COLLECTION_NAME; // persistence
location
```

We describe commercetools Message schema using joi and define a helper function - **getValidationError** - to look for errors in the messages we receive. We only want to persist valid order messages and joi will perform data validation for us. We are doing very simple validation here but joi is capable of doing much more if you wish.

```
const orderSchema = Joi.object({
  createdAt: Joi.string().required(),
  id: Joi.string().required(),
  lastModifiedAt: Joi.string().required(),
  order: Joi.object(),
  orderId: Joi.string(),
  resource: Joi.object().required(),
  resourceVersion: Joi.number().required(),
  sequenceNumber: Joi.number().required(),
  type: Joi.string().required(),
  version: Joi.number().required(),
})
   .or('order', 'orderId') // order on create/delete; orderId on
update
  .unknown(); // allow top level unknown keys

const getValidationError = async (order) => {
  try {
    const value = await orderSchema.validateAsync(order);
    return value.error !== undefined;
  }
  catch (err) {
    return err.message;
  }
}
```

Finally, we have the exported cloud function itself, **orderArchive**. There are two types of Google Cloud Functions: HTTP functions and background functions. We will use a background function as they can be automatically executed when a message is received on a Pub/Sub topic. A background function takes three parameters:

- `message` - an object containing the Cloud Pub/Sub message
- `context` - an object containing meta-data (e.g., eventId, eventType, etc)
- `callback` - a function signalling completion of the function's execution

The function does three things. First, it extracts content from the message parameter. Pub/Sub uses base64 to encode the data it publishes so we need to decode it and convert it to a JavaScript object. Note that this might fail if we have bad data so everything is wrapped in a try-catch block so we can log errors if they occur.

Second, we perform data validation to ensure we received a commercetools Message that includes order data. The call to **getValidationError** returns false if we are safe to save the data; otherwise, it will return an error message that we log.

Third, we use Firestore to save the order to our project in the collection specified in our environment variables.

```
exports.orderArchive = async (message, context, callback) => {
  const { eventId } = context || {};
  try {
    // 1. extract the order from the pubsub message
    const { data } = message || {};
   const order = JSON.parse(Buffer.from(data, 'base64').toString());

    // 2. validate the order; noop if its invalid
    const validationError = await getValidationError(order);
    if (!validationError) {

      // 3. persist the order to firestore
      const firestore = new Firestore({ projectId: PROJECTID });
      const result = await firestore
        .collection(COLLECTION_NAME)
        .add(order);

      callback(null, 'Success');
      console.log({ message: 'orderArchive success', eventId });
      firestore.terminate();

    } else {
      // function successful but payload was bad
      callback(null, `Order Invalid: ${validationError}`);
      console.log({
        message: `orderArchive invalid: ${validationError}`,
        order: JSON.stringify(order),
        eventId,
      });
    }
```

```
  } catch (error) {
    const { message = 'Unknown error', stack } = error;
    console.error({
      eventId,
      message: `orderArchive error: ${message}`,
      stack,
    });
    callback(error); // function unsuccessful
  }
};
```

Step D: Configure and Deploy the Cloud Function.

Before we deploy or run the function, we need to do some configuration. To run locally, we need to set three environment variables: GOOGLE_APPLICATION_CREDENTIALS, PROJECTID and COLLECTION_NAME.

To set up your credentials, follow the Creating a service account instructions from Google. This will walk you through five steps to create a service account and download the credentials to your local system.

We can then use a .env file to set the variables. If you don't remember your project ID, you can retrieve it at https://console.cloud.google.com/home/dashboard. Feel free to change the collection name but we will use "ct-orders". Create a file named `.env` in the root of the repo using this as your guide for its content:

```
GOOGLE_APPLICATION_CREDENTIALS="/PATH/TO/YOUR/DOWNLOADED/CREDENTIALS"
PROJECTID="YOUR-PROJECT-ID"
COLLECTION_NAME="ct-orders"
```

When we deploy the function to GCP, we will need a way to set the PROJECTID and COLLECTION_NAME there too. We can specify these in the deployment.yaml file so go ahead and update that now too:

```
PROJECT_ID: YOUR-PROJECT-NAME
COLLECTION_NAME: ct-orders
```

Note that we do not need to specify credentials in the deployment.yaml because the cloud function will inherit the permissions it needs once deployed to GCP.

We can manually perform a basic integration test at this point. The repo makes this fairly simple if you're using VSCode, see the debug configuration for it in launch.json.

Open index.js, go to VSCode's Run tab, select "debug:orderArchive" from the list of configurations, and click the Start Debugging icon. If you're not using VSCode, you can still run the function by executing `npm start` from the command line but you will need to edit the start script to pass in your environment variables; I suggest using a tool like cross-env if you want to go that route.

Once the function is running locally, you can POST to it. If you want to create your own sample, remember that data in a Pub/Sub message is base64 encoded; here is a test body you can use:

```
{
  "data": {
    "data":
"ewogICJwcm9qZWN0S2V5IjogImZ0Zy1kZXZlbG9wbWVudCIsCiAgInR5c-
GUiOiAiT3JkZXJDcmVhdGVkIiwKICAibGFzdE1vZGlmaWVkQXQiOiAiMjAyMC0x-
MC0xMlQyMToxNDo0NC40ODNaIiwKICAib3JkZXIiOiB7CiAgICAidGVzdCI6IHRy-
dWUKICB9LAogICJub3RpZmljYXRpb25UeXBlIjogIk1lc3NhZ2UiLAogICJjcmVh-
dGVkQXQiOiAiMjAyMC0xMC0xMlQyMToxNDo0NC40ODNaIiwKICAic2VxdWVuY2VOd-
W1iZXIiOiAxLAogICJyZXNvdXJjZSI6IHsKICAgICJ0ZXN0IjogdHJ1ZQogIH0s-
CiAgInJlc291cmNlVXNlclByb3ZpZGVkSWRlbnRpZmllcnMiOiB7fSwKICAicmVz-
b3VyY2VWZXJzaW9uIjogMSwKICAidmVyc2lvbiI6IDEsCiAgImlkIjogInRlc3Qx-
Igp9"
  }
}
```

The function should return `{ message: 'orderArchive success', eventId: undefined }` and you should see a new document saved in the Cloud Firebase console if everything went according to plan.

Assuming your integration test was successful, we should deploy the function. There are a few GCP prerequisites we need to fulfill; these only need to be done once in your GCP project:

1. Make sure that billing is enabled for your Google Cloud project by following the instructions at Confirm billing is enabled on a project.

2. Enable the Cloud Functions and Cloud Build APIs by following the Before you begin instructions.

3. Make sure the APIs are enabled and the Cloud Functions Developer role is enabled for your service account by following the Open the Cloud Build Settings page link shown in the instructions. If there was an issue enabling the APIs, you can follow the steps to enable that, and then return the Build Settings to enable the role.

You can deploy locally using the gcloud CLI from the root of the repo; substitute PROJECTID with the ID of your project and update the region if you made a different location selection in Step B:

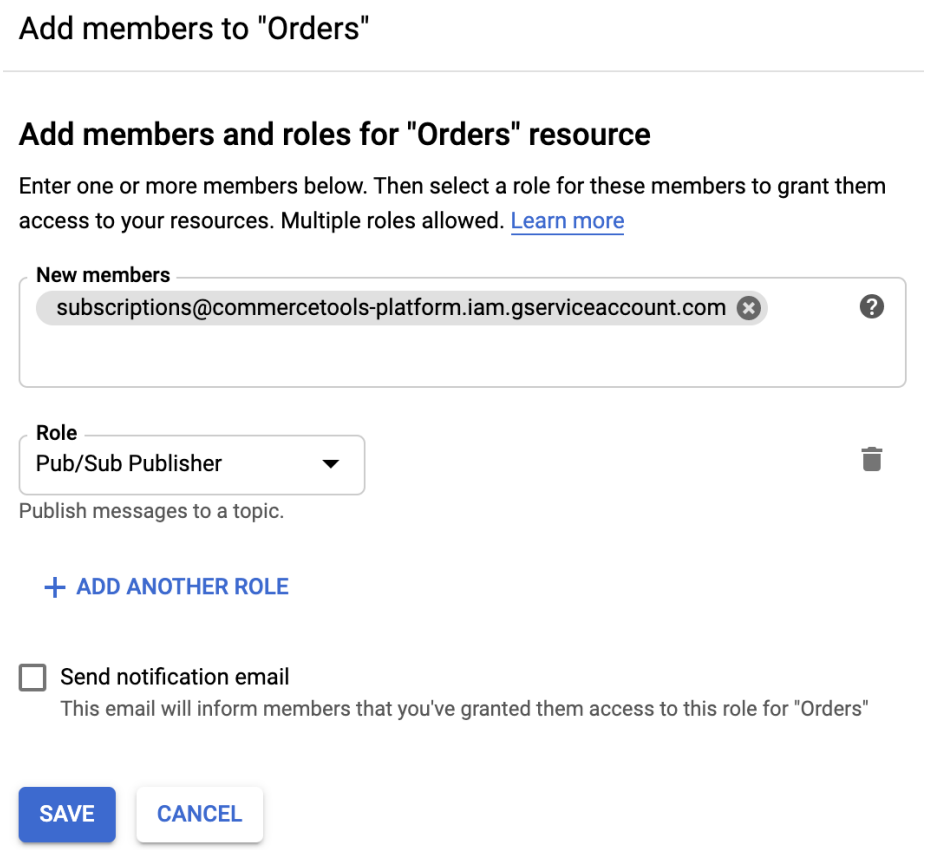
```
 gcloud functions deploy orderArchive --project PROJECTID --region
us-west3   --env-vars-file   ./deployment.yaml   --runtime   nodejs12
--trigger-topic Orders
```

The same command is available in the package.json deploy script. So, assuming you update PROJECTID there, you can execute the same command by executing `npm run deploy` from the command line.

Step E: Set up the GCP Pub/Sub Topic.

When we deployed our Cloud Function in the previous step, GCP automatically set up the associated Orders Topic for us based on the --trigger-topic Orders parameter we provided. You can confirm that by going to https://console.cloud.google.com/cloudpubsub/topic/list. If a Topic called "Orders" is not there, we can easily create one by clicking the "Create Topic" button you should see at the top of the Topics page. Enter "Orders" as the Topic ID in the dialog box that pops up. Leave the "Use a customer-managed encryption key" unchecked.

We need to grant commercetools permission to publish to our new topic. Click on the "Orders" topic to see its details. Make sure the Info Panel is shown; click the "SHOW INFO PANEL" button in the upper-right if the panel is hidden. In the Info Panel's Permissions tab, click the "ADD MEMBER" button. In the form provided, add "subscriptions@commercetools-platform.iam.gserviceaccount.com" as the **New members** and "Pub/Sub Publisher" as the **Role** as shown below.



Click Save and GCP is now ready to receive messages from a commercetools Subscription.

You can check to ensure that our Cloud Function is associated with our Orders topic by scrolling down to the Subscriptions at the bottom of the Order topic's detail page. You should see an entry for "gcf-orderArchive-us-west3-Orders". The region name may be different for you. If that entry is missing, you will need to revisit the Cloud Function deploy in the previous step.

Let's move to the commercetools side next.

Step F: Create a commercetools project.

If you already have a commercetools project you can skip this step. If not, there is good news: you can easily sign up for a risk-free, fully-functional 60 day commercetools trial. The trial period does introduce a few limits, like the total number of products you can define, but the feature set is rich.

Go to https://commercetools.com/free-trial and fill out the form to get an email with instructions for creating your trial organization and initial project. The process is quite fast because commercetools automates all the work behind the scenes to provision cloud resources for you. Note the key you used for your project as it will be used in Step G. Once you have your first project in place, proceed to the next step.

Step G: Load commercetools sample data.

To test our Subscription, we need to be able to create and modify Order resources on the commercetools platform. If you already have data in your commercetools project or are comfortable creating the resources we need using tools like the Merchant Center, IMPEX or the HTTP API, then you can skip this step.

Alternatively, commercetools provides an open source project to make loading sample data easy. If you're comfortable running open source tools, you can follow the steps in the Sunrise Data README; if not, here is what you should do:

1. Open a command line (our examples use bash) and issue these three commands to clone and initialize the commercetools-sunrise-data open source repository:

```
$ git clone
https://github.com/commercetools/commercetools-sunrise-data.git
$ cd commercetools-sunrise-data/
$ npm install
```

2. The commercetools-sunrise-data application needs some configuration so it knows what project to load the data into and has the credentials it needs to perform its work. Here are the steps:

a. Login to the Merchant Center and then navigate to Settings -> Developer Settings from the left navigation.
b. Click “Create new API Client”
c. For field **Name**, enter: admin-client
d. For field **Scopes**, select: Admin client
e. Click “Create API Client”
f. Note all the information provided by the Merchant Center as we will use them in the next step.

Now that we have the configuration details we need, we can create a `.env` file for the commercetools-sunrise-data application to leverage. Create a new file called .env at the root of your commercetools-sunrise-data directory. It should have the following entries; replace the generic values with information you captured in the previous step. If you lost your configuration details, you can perform the previous step again and create a new API Client without harm:

```
CTP_PROJECT_KEY = <your project key>
CTP_CLIENT_ID = <your client ID>
CTP_CLIENT_SECRET = <your client secret>
CTP_API_URL = <your apiUrl> (i.e., api.commercetools.com)
CTP_AUTH_URL = <your authUrl> (i.e., auth.commercetools.com)
```

You are now ready to load data. Assuming all the previous steps were successfully followed, a single command will load data for you. Note that this command will **replace all data in the project!** If you need to retain existing data, see further instructions in the README.md.

```
$ npm run start
```

Step H: Set up commercetools Subscription.

We can Create a Subscription in commercetools by sending a POST to the `/{projectKey}/subscriptions` endpoint. The endpoint accepts a SubscriptionDraft which allows you to specify the following:

- `key` - String - Optional - User-specific unique identifier for the subscription. We will set this to “gcp-order-subscription”
- `destination` - Destination - The Message Queue into which the notifications are to be sent.
  A wide variety of message queues are supported; we will use GCP by setting: `{ “type”: “GoogleCloudPubSub”, “projectId”: “PROJECTID”, “topic”: “Orders” }`

- `messages` - Array of MessageSubscription - Optional - The messages to be subscribed to.
  We will subscribe to all messages for the Order resource by setting:
  `[{ "resourceTypeId": "order" }]`
- `changes` - Array of ChangeSubscription - Optional - The change notifications to be subscribed to.
  We will leave this undefined.
- `format` - Format - Optional - The format in which the payload is delivered.
  We will use the default which is Platform Format.

Based on the above, here is what we want to POST:

```
{
  "key": "gcp-order-subscription",
  "destination": {
    "type": "GoogleCloudPubSub",
    "projectId": "PROJECTID",
    "topic": "Orders"
  },
  "messages": [
    {
      "resourceTypeId": "order"
    }
  ]
}
```

We can use IMPEX to set this up. Go to the API Playground and login. In the **Endpoint** field, select "Subscriptions". In the **Command** field, select "Create". In the **Payload** field, paste the JSON from above. Make sure you change PROJECTID to be the ID of your GCP project. Click the GO button and IMPEX should reply with something similar to:

```
{
  "id": "3d678687-0519-4c1a-9120-4b518b7a92d4",
  "version": 1,
  "createdAt": "2020-10-10T20:33:35.039Z",
  "lastModifiedAt": "2020-10-10T20:33:35.039Z",
  "lastModifiedBy": {
    "isPlatformClient": true
  },
  "createdBy": {
    "isPlatformClient": true
  },
  "destination": {
    "type": "GoogleCloudPubSub",
    "projectId": "PROJECTID",
    "topic": "Orders"
  },
```

```
“changes”: [],
  “messages”: [
    {
      “resourceTypeId”: “order”
    }
  ],
  “format”: {
    “type”: “Platform”
  },
  “status”: “Healthy”,
  “key”: “gcp-order-subscription”
}
```

To alternatively use the HTTP API, we can take advantage of tools like Postman or curl. If you are familiar with Postman, commercetools provides a repository containing Postman collections for the platform. We will show examples using curl. If you’re using Windows locally, this guide from Zendesk may be useful as it documents modifications you may need to make.

We can use curl in two steps. First, we need an authorization token. Run the following from the command line, substituting AUTH_HOST, CLIENT_ID, SECRET and PROJECT_KEY with data we noted in Step G:

```
curl https://AUTH_HOST/oauth/token \
--basic --user “CLIENT_ID:SECRET” \
-X POST \
-d               “grant_type=client_credentials&scope=manage_
project:PROJECT_KEY”
```

Second, we use the returned `access_token` to provide authorization when performing the POST. Run the following from the command line, substituting ACCESS_TOKEN with the `access_token` returned from the last curl request, API_HOST and PROJECT_KEY with the commercetools project key we noted in Step G, and PROJECTID with your GCP project ID:

```
curl -sH “Authorization: Bearer ACCESS_TOKEN” \
-H ‘content-type: application/json’ \
-d      ‘{“key”:      “gcp-order-subscription”,”destination”:
{“type”:”GoogleCloudPubSub”,     “projectId”:     “PROJECTID”,
“topic”: “Orders”}, “messages”: [{“resourceTypeId”: “order”}]}’
\
https://API_HOST/PROJECT_KEY/subscriptions
```

Your response will be similar to the JSON response from IMPEX.

Step I: Test commercetools/GCP Integration.

We are now ready to test our integration. All create, update and delete operations on orders should now cause commercetools to publish a message to our GCP Pub/Sub Topic which will trigger our Cloud Function to execute which results in a new document being added to our Firestore ct-orders collection.

Let's test it out using IMPEX. Go to the API Playground and login. If you loaded the sample data in Step G, we can create a cart using one of the SKUs we loaded. Set the **Endpoint** field to "Carts", the **Command** field to "Create", the **Payload** field to:

```
{
  "currency": "USD",
  "country": "US",
  "lineItems": [{
    "sku": "M0E20000000DZWJ"
  }],
  "shippingAddress": {
    "country": "US"
  }
}
```

Hit GO and IMPEX will respond with a Cart object. Copy the `id` and `version` fields from the response so we can use them to create an Order. We will Create an Order from Cart in IMPEX by setting the **Endpoint** field to "Orders", the **Command** field to "Create", and the **Payload** field to the two fields you copied:

```
{
  "id": "ID-FROM-CART",
  "version": VERSION-FROM-CART
}
```

Hit GO and IMPEX will respond with an Order object. Copy the `id` of the cart so we can find it on GCP next.

Go to Firestore at https://console.cloud.google.com/firestore/data/ct-orders and click on the filter icon next to the ct-orders column header. Set the **Choose a field to filter** by field to "order.id", set the **Add a condition field** to "(==) equal to" and set the **String** field to the `id` from the commercetools Order. Hit the Apply button and you should see the order we created.

If you don't see the order, you can troubleshoot by reviewing the Cloud Function logs. Go to https://console.cloud.google.com/functions/list and click the orderArchive function to see its details. Click the **Logs** tab to see its output sorted by date and time. If nothing is there, it likely means your Subscription configuration is not working so review Step H. If the cloud function received a message but errored, you will see the issue reported in the list. If everything worked as expected, you will see a success message like, "`{ message: 'orderArchive success', eventId: '1631128022312033' }`".

If you had success with order creation, we can perform an example update next. In IMPEX, we can Update the Order by ID using an UpdateAction of Change OrderState. The payload we send changes the `orderState` field; use the `version` from the Order we created above:

```
{
  "version": 1,
  "actions": [{
    "action": "changeOrderState",
    "orderState": "Confirmed"
  }]
}
```

In IMPEX, set the **Endpoint** field to "Orders", the **Command** field to "Update", set the **Payload** to the JSON above and the **Resource ID** to the `id` from the Order we created above. Hit GO and IMPEX will respond with the updated order. You should see two changes: the `version` will be incremented and the `orderState` will be "Confirmed".

Head over to your GCP console and check Firestore again. The subscription message we get for an update is different from a create. A create message has a `type` of "OrderCreated" and includes the complete order object in field `order`. An update message has a type of "OrderStateChanged" and does not include the entire order object. Instead, only the changed fields are communicated. In our example, we get fields: `orderId`, `orderState`, and `oldOrderState`. So, we can use orderId to find our update message. Hit the filter button and this time set the **Choose a field to filter** by to "orderId" to see the update.

Finally, we can test a delete. In IMPEX, set the **Endpoint** field to "Orders", the **Command** field to "Delete", the **Resource ID** to the `id` from the Order we created above, and the **Resource version** to the `version` returned from the update request. Hit GO and IMPEX will respond with the deleted order. In Firebase, the document archived will have a `type` of "OrderDeleted" and will contain an `order` field, like we saw when we initially created the order. So, a filter on order.id should now show two records, one for the create and one for the delete.

Note that all the IMPEX requests made above can alternatively be performed using the commercetools HTTP API.

Feel free to experiment more. Both GCP and commercetools know how to autoscale so in a production environment, as orders come in, compute resources will be provisioned dynamically to handle the load and archive all order updates as they are published.

We are done! In this example use case, we limited ourselves to archiving all order messages. There are, of course, many other things you could do based on your needs: send customers update emails in response to `orderState` changes, feed orders into downstream business intelligence systems, generate alerts based on updates using your business rules, etc. We hope this paper gives you the baseline information you need to innovate using commercetools and Subscriptions!

Step J: Provide you with resources if you need help.

We travelled quite a bit of ground covering Subscriptions and showing you the power they provide. If you have q
Questions or need additional help, Fearless Technology Group (FTG) is available to assist you. Shoot us an email at contactus@fearlesstg.com so we can lend a hand.

Additional Help

As you consider all your customization options, FTG and commercetools are here to assist you.

About Fearless Technology Group

Fearless Technology Group (FTG) helps retailers modernize their technology architecture, solve critical business problems, and capitalize on business opportunities in an evolving landscape. FTG is both a commercetools and Google Cloud Platform Partner. Contact us at contactus@fearlesstg.com or 720-432-9068.

About commercetools

commercetools is a next-generation software technology company that offers a true cloud commerce platform, providing the building blocks for the new digital commerce age. Our leading-edge API approach helps retailers create brand value by empowering commerce teams to design unique and engaging digital commerce experiences everywhere – today and in the future. Our agile, componentized architecture improves profitability by significantly reducing development time and resources required to migrate to modern commerce technology and meet new customer demands. It is the perfect starting point for customized microservices.

Europe - HQ
commercetools GmbH
Adams-Lehmann-Str. 44
80797 Munich, Germany
Tel. +49 (89) 99 82 996-0
info@commercetools.com

Americas
commercetools, Inc.
324 Blackwell, Suite 120
Durham, NC 27701
Tel. +1 212-220-3809
mail@commercetools.com

www.commercetools.com
Munich - Berlin - Jena - Amsterdam - London - Durham NC - Singapore - Melbourne