Local development for Cloud Run with Docker Compose

Author(s): @grayside (https://github.com/grayside), Published: 2019-05-21

Adam Ross | Developer Programs Engineer | Google

Contributed by Google employees.

This tutorial shows how to use Docker Compose (https://docs.docker.com/compose/overview/) to streamline your local development environment for Cloud Run (/run).

Overview

Services running on Cloud Run are running in containers, so you probably want to identify how to use or build a local container toolchain that can work with Cloud Run and integrate with other Google Cloud products.

The first thing to know: you do not have to use Docker locally. Cloud Build can build your container images remotely, and your services can be built to work outside a container. Deciding whether the practice of containerizing services for local development is outside the scope of this tutorial. Instead, we will assume that you want to use containers as much as possible.

Docker Compose is a wonderful utility to build out a locally containerized workflow and align your team on common practices. It allows many of the Docker management details to be pulled out of your head (and shell scripts) and captured in declarative configuration files for your version control system.

Let's explore how we can use it to build a local development workflow for your Cloud Run project.

This tutorial builds on some of the details in the Local testing documentation (/run/docs/testing/local).

Service directory structure

Let's imagine a service written in Go and ready for production on Cloud Run:

```
.
├── .dockerignore
├── .gcloudignore
├── .git
├── Dockerfile
├── go.mod
└── main.go
```

Our goal is now to add the YAML configuration files that Docker Compose will use to create, configure, and build the local container images for this service.

Your local, basic setup: `docker-compose.yml`

This foundational configuration file demonstrates how to configure your Cloud Run service for local use. It does not attempt to replicate Knative beyond some environment variables to approximate the Container runtime contract (/run/docs/reference/container-contract).

```
# docker-compose.yml
version: '3'

services:
 app:
   build: .
   image: sample-app:local
   ports:
     # Service will be accessible on the host at port 9090.
     - "9090:${PORT:-8080}"
   environment:
     # /run/docs/reference/container-contract
     PORT: ${PORT:-8080}
     K_SERVICE: sample-app
     K_REVISION: 0
     K_CONFIGURATION: sample-app
```

What can you do with this?

- Create and start all configured services with **`docker-compose up`**.
- Build your container images for local use with **`docker-compose build`**.

For more, check out the Docker Compose CLI documentation (https://docs.docker.com/compose/reference/overview/).

Using Google Cloud APIs from your local container

When using the official Google Cloud client libraries on Cloud Run, authentication to other Google Cloud services is automatically handled through the service account provisioned into your Cloud Run service. No further steps are required.

When running your containerized services locally, you can take advantage of this same library capability by injecting service account credentials into your container at runtime.

To authenticate your local service with Google Cloud, do the following:

1. Follow the steps in the authentication documentation (/docs/authentication/getting-started) to create a service account and download service account keys to your local machine.
2. Configure your container and service to use these keys for authentication.

   Files in this tutorial named docker-compose.[topic].yml rely on an inheritance model built into `docker-compose` for multiple configuration files (https://docs.docker.com/compose/extends/#multiple-compose-files). The way you choose to split up these configurations will be dependent on your needs. The approach here is driven by the sequence of topics:

```
# docker-compose.access.yml
# Usage:
#   export GCP_KEY_PATH=~/keys/project-key.json
#   docker-compose -f docker-compose.yml -f docker-compose.access.
version: '3'
services:
  app:
    environment:
      # /docs/authentication/production
      GOOGLE_APPLICATION_CREDENTIALS: /tmp/keys/keyfile.json
    volumes:
      # Inject your specific service account keyfile into the cont
      - ${GCP_KEY_PATH}:/tmp/keys/keyfile.json:ro
```

   The `$GCP_KEY_PATH` environment variable is set in your local machine—outside the container—to pass the contents of your key file into the container.

3. Now you can start your service with a command such as the following:

```
export GCP_KEY_PATH=~/keys/project-key.json
docker-compose -f docker-compose.yml -f docker-compose.access.yml
```

The client libraries will make API calls with the service account credentials. Access to other services will be limited by the roles and permissions associated with that account.

Shipping releases to Container Registry

This section provides guidance on interacting with Container Registry (/container-registry), a Docker Container registry used as the source of container images deployed to Cloud Run.

Using the configuration inheritance described above to *override* a setting allows us to locallly build and push a release artifact to Container Registry.

```
##
# docker-compose.gcp.yml
#
# Usage:
#   export DOCKER_IMAGE_TAG=$(git rev-parse --short HEAD)
#   docker-compose -f docker-compose.yml -f docker-compose.gcp.yml
##
version: '3'

services:
 app:
   image: gcr.io/my-project-name/sample-app:${DOCKER_IMAGE_TAG:-latest
```

This image name override helps differentiate images built for local use from images built to push to gcr.io.

You may prefer to use Cloud Build (/cloud-build) to build your images without tying up local resources.

Build a container image for each `docker-compose` service

```
docker-compose \
-f docker-compose.yml \
```

```
-f docker-compose.gcp.yml \
build
```

Build a container image for the specified `docker-compose` service

```
docker-compose \
-f docker-compose.yml \
-f docker-compose.gcp.yml \
build app
```

Push your container images to Container Registry

First you must authenticate the Docker CLI with Container Registry (/run/docs/building/containers#building_locally_and_pushing_using_docker):

```
docker-compose \
-f docker-compose.yml \
-f docker-compose.gcp.yml \
push
```

Pull your published container images for local use

If you want to explore your published container image, such as getting a closer look at the Docker image that your production service is currently running, you may pull the image down from Container Registry. This also requires Docker CLI authentication.

Note: Your services will not be updated to this pulled image automatically; you may need to restart or remove the existing containers.

```
export DOCKER_IMAGE_TAG=[PRODUCT_TAG]
docker-compose \
-f docker-compose.yml \
```

```
-f docker-compose.gcp.yml \
pull
```

Connect your local service to Cloud SQL

It is common for local development to use a local database server. However, if you need to access a Cloud SQL instance, such as for remote administration, you can use the Cloud SQL Proxy (/sql/docs/mysql/sql-proxy).

The Cloud SQL Proxy has an officially supported containerized solution (/sql/docs/mysql/connect-docker).

Let's adapt that documentation for use with our **`docker-compose`** configuration. We will use environment variables to configure the proxy:

```
# docker-compose.sql.yml
#
# Usage:
#   export GCP_KEY_PATH=~/keys/project-sql-key.json
#   export CLOUDSQL_CONNECTION_NAME=project-name:region:instance-name
#   export CLOUDSQL_USER=root
#   export CLOUDSQL_PASSWORD=""
#   docker-compose -f docker-compose.yml -f docker-compose.sql.yml
version: '3'

services:
 app:
   environment:
     # These environment variables are used by your application.
     # You may choose to reuse your production configuration as implie
     # but an alternative database instance and user credentials is re
     - CLOUDSQL_CONNECTION_NAME
     - CLOUDSQL_USER
     - CLOUDSQL_PASSWORD
   volumes:
     # Mount the volume for the cloudsql proxy.
     - cloudsql:/cloudsql
   depends_on:
    - sql_proxy
```

```
 sql_proxy:
   image: gcr.io/cloudsql-docker/gce-proxy:1.19.1
   command:
     - "/cloud_sql_proxy"
     - "-dir=/cloudsql"
     - "-instances=${CLOUDSQL_CONNECTION_NAME}"
     - "-credential_file=/tmp/keys/keyfile.json"
   # Allow the container to bind to the unix socket.
   user: root
   volumes:
     - ${GCP_KEY_PATH}:/tmp/keys/keyfile.json:ro
     - cloudsql:/cloudsql

volumes:
 # This empty property initializes a named volume.
 cloudsql:
```

The service account used by the Cloud SQL Proxy must include the Project Viewer, Cloud SQL Viewer, and Cloud SQL Client roles. Do not whitelist your IP address with the MySQL instance.

Similar to the **`docker-compose.access.yml`** example, this file layers on top of your **`docker-compose.yml`**. You can stack all three together to start your service with full Google Cloud access:

```
docker-compose \
-f docker-compose.yml \
-f docker-compose.access.yml \
-f docker-compose.sql.yml \
up
```

This configuration will start up two containers: one for your service and one for the Cloud SQL Proxy. They will hand off interactions with the Cloud SQL database via the shared **`cloudsql`** volume.

Submit a tutorial

Share step-by-step guides

SUBMIT A TUTORIAL (/COMMUNITY/TUTORIALS/WRITE)

Request a tutorial

Ask for community help

SUBMIT A REQUEST (HTTPS://GITHUB.COM/GOOGLECLOUDPLATFORM/COMMUNITY/ISSUES?Q=IS%3AOPEN+IS%3AISSUE+LABEL%3A%22TUTORIAL+REQUEST%22)

View tutorials

Search Google Cloud tutorials

VIEW TUTORIALS (/DOCS/OPEN-TUTORIALS)

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License (http://creativecommons.org/licenses/by/4.0/), and code samples are licensed under the Apache 2.0 License (http://www.apache.org/licenses/LICENSE-2.0).

For details, see our Site Policies (https://developers.google.com/terms/site-policies). Java is a registered trademark of Oracle and/or its affiliates.