Week 4, Cloud Application Development

Azamat Serek, PhD, Assist.Prof.

Cloud run functions

Cloud Run functions is a serverless execution environment for building and connecting cloud services. With Cloud Run functions you write simple, single-purpose functions that are attached to events emitted from your cloud infrastructure and services. Your function is triggered when an event being watched is fired, or by an HTTP request.

This page shows how to create and deploy an HTTP function using the Google Cloud console. This page is based on Node.js, but the process is similar for all of the runtimes.

Example

The example in this quickstart uses the following Node.js function, which returns a message when triggered by an HTTP request:

```
const functions = require('@google-cloud/functions-framework');

functions.http('helloHttp', (req, res) => {
 res.send(`Hello ${req.query.name || req.body.name || 'World'}!`);
});
```

Before you begin

1. In the Google Cloud console, on the project selector page, select or create a Google Cloud project.

   ★ **Note**: If you don't plan to keep the resources that you create in this procedure, create a project instead of selecting an existing project. After you finish these steps, you can delete the project, removing all resources associated with the project.

   Go to project selector

2. Make sure that billing is enabled for your Google Cloud project.

3. Enable the Cloud Functions, Cloud Build, Artifact Registry, Cloud Run, Logging, and Pub/Sub APIs.

   Enable the APIs

Create a function

1. Open the Functions Overview page in the Google Cloud console:

   Go to the Cloud Run functions Overview page

   Make sure that the project for which you enabled Cloud Run functions is selected.

2. Click **Create function**.

3. Name your function, for example, `function-1`.

4. Select a **Region** in which to deploy your function.

5. In the **HTTPS** field under **Trigger**, leave **Require authentication** selected. This is the default setting.

   The other option, **Allow unauthenticated invocations**, lets you reach the function without authentication. This is useful for testing, but we don't recommend using this setting in production unless you are creating a public API or website. Further, it might not work for you, depending on your corporate policy settings. See Authenticating for invocation for details on how to invoke a function that requires authentication.

6. Click **Next**.

7. In the **Source code** field, select **Inline editor**. In this exercise, you use the default function provided in the editor.
8. Use the **Runtime** dropdown to select the desired runtime. This example uses `nodejs20`.

Deploy the function

1. At the bottom of the page, click **Deploy**.
2. After clicking **Deploy**, the Google Cloud console redirects to the **Function details** page.

While the function is being deployed, the icon next to it is a small spinner. After the function finishes deploying, the spinner turns to a green check mark.

Test the function

To test the function after it has finished deploying:

1. Open the **Testing** tab.
2. Scroll down to the **CLI test command** field.
3. Click **Run in Cloud Shell**.

   A Cloud Shell window opens at the bottom of your screen, displaying the `curl` command from the **Testing** tab. You might be prompted to authorize Cloud Shell.
4. To execute the `curl` command that is displayed in your Cloud Shell window, press `Return`.

   Your "Hello world" message is displayed.

Exercise

Do it yourself

https://cloud.google.com/functions/docs/console-quickstart

Write Cloud Run functions

Send feedback

Cloud Run functions supports writing source code in a number of programming languages. The language runtime you choose and the type of function you want to write determine how to structure your code and implement your function. This page provides an overview of the types of Cloud Run functions and expectations for source code.

Types of Cloud Run functions

There are two types of Cloud Run functions:

- **HTTP functions**, which handle HTTP requests and use HTTP triggers. See Write HTTP functions for information about implementing HTTP functions.
- **Event-driven functions**, which handle events from your cloud environment and use event triggers as described in Cloud Run functions triggers. See Write event-driven functions for information about implementing event-driven functions.

Source directory structure

In order for Cloud Run functions to locate your function definition, each language runtime has requirements for structuring your source code. The basic directory structure for a function in each runtime is shown below.

Node.js | Python | Go | Java | C# | Ruby | PHP

The basic directory structure for Python functions is as follows:

```
.
├── main.py
└── requirements.txt
```

Cloud Run functions loads source code from a file named `main.py` at the root of your function directory. Your main file must be named `main.py`.

Your `requirements.txt` file must include the Functions Framework for Python as a dependency:

Your `requirements.txt` file must include the Functions Framework for Python as a dependency:

```
functions-framework==3.*
```

The code in your `main.py` file must define your function entry point and can import other code and external dependencies as normal. The `main.py` file can also define multiple function entry points that can be deployed separately.

If you are thinking of grouping multiple functions into a single project, be aware that every function may end up sharing the same set of dependencies. However, some of the functions may not need all of the dependencies.

Where possible, we recommend splitting up large multi-function codebases and putting each function in its own top-level directory as shown above, with its own source and project configuration files. This approach minimizes the number of dependencies required for a particular function, which in turn reduces the amount of memory your function needs.

Function entry point

Function entry point

Your source code must define an *entry point* for your function, which is the particular code that is executed when the Cloud Run function is invoked. You specify this entry point when you deploy your function.

How you define the entry point depends on the language runtime you use. For some languages, the entry point is a function, whereas for others the entry point is a class. To learn more about defining entry points and implementing Cloud Run functions in different languages, see Write HTTP functions and Write event-driven functions.

Exercise

Do this yourself

https://cloud.google.com/functions/docs/writing#directory-structure-python

Write HTTP functions

In Cloud Run functions, you use HTTP functions when you want to invoke a function via an HTTP(S) request. To allow for HTTP semantics, HTTP function signatures accept HTTP-specific arguments.

```python
import functions_framework

# Register an HTTP function with the Functions Framework
@functions_framework.http
def my_http_function(request):
  # Your code here

  # Return an HTTP response
  return 'OK'
```

In Python, you register an HTTP handler function with the Functions Framework for Python. Your HTTP handler function must accept a Flask request object as an argument and return a value that Flask can convert into an HTTP response object.

The function entry point is the name of the handler function registered with the Functions Framework. In this example, the entry point is `my_http_function`.

HTTP requests and responses

HTTP functions accept the HTTP request methods listed on the page HTTP triggers. Your HTTP handler can inspect the request method and perform different actions based on the method.

Your function must send an HTTP response. If the function creates background tasks (such as with threads, futures, JavaScript `Promise` objects, callbacks, or system processes), you must terminate or otherwise resolve these tasks before sending an HTTP response. Any tasks not terminated before the HTTP response is sent might not be completed, and might cause undefined behavior.

See HTTP triggers for more information about HTTP functions and associated options.

Handling CORS

Cross-Origin Resource Sharing (CORS) is a way to let applications running on one domain access resources from another domain. For example, you might need to let your domain make requests to the Cloud Run functions domain to access your function.

If CORS is not set up properly, you might see errors like the following:

```
XMLHttpRequest cannot load https://YOUR_FUNCTION_URL.
No 'Access-Control-Allow-Origin' header is present on the requested resource.
Origin 'https://YOUR_DOMAIN' is therefore not allowed access.
```

To allow cross-origin requests to your function, set the `Access-Control-Allow-Origin` header as appropriate on your HTTP response. For preflighted cross-origin requests, you must respond to the preflight `OPTIONS` request with a `204` response code and additional headers.

```python
@functions_framework.http
def cors_enabled_function(request):
    # For more information about CORS and CORS preflight requests, see:
    # https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request

    # Set CORS headers for the preflight request
    if request.method == "OPTIONS":
        # Allows GET requests from any origin with the Content-Type
        # header and caches preflight response for an 3600s
        headers = {
            "Access-Control-Allow-Origin": "*",
            "Access-Control-Allow-Methods": "GET",
            "Access-Control-Allow-Headers": "Content-Type",
            "Access-Control-Max-Age": "3600",
        }

        return ("", 204, headers)

    # Set CORS headers for the main request
    headers = {"Access-Control-Allow-Origin": "*"}

    return ("Hello World!", 200, headers)
```

Exercise

Do this yourself

https://cloud.google.com/functions/docs/writing/write-http-functions

References

1. https://cloud.google.com/functions?hl=ru