This page shows you how to set up and use Stackdriver Profiler. You download a sample Go program, run it with profiling enabled, and then use the Profiler interface to explore the captured data.

1. Sign in (https://accounts.google.com/Login) to your Google Account.

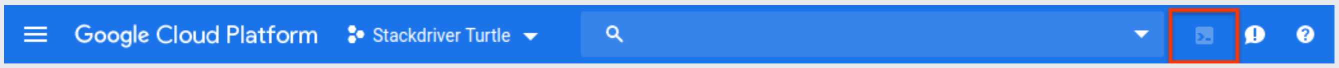
   If you don't already have one, sign up for a new account (https://accounts.google.com/SignUp).
2. In the Cloud Console, on the project selector page, select or create a Cloud project.

   ★ **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.

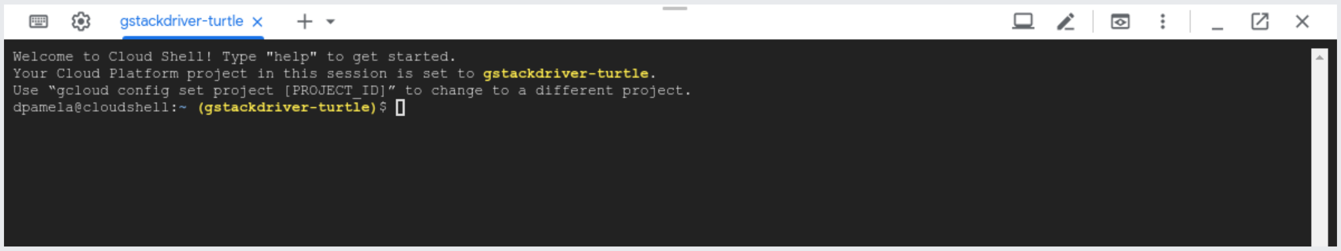
   Go to the project selector page (https://console.cloud.google.com/projectselector2/home/dashboard)
3. Enable the Stackdriver Profiler API.

   Enable the API (https://console.cloud.google.com/flows/enableapi?apiid=cloudprofiler.googleapis.com&redirect=https://console.cloud.google.com)

At the top of the Google Cloud Console page for your project, click **Activate Cloud Shell**:



A Cloud Shell session opens inside a new frame at the bottom of the console and displays two messages and a command-line prompt. The first message lists the Google Cloud project for your Cloud Shell session. The second message tells you how to change the session project. It can take a few seconds for the shell session to be initialized:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to gstackdriver-turtle.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
dpamela@cloudshell:~ (gstackdriver-turtle)$
```

The sample program, `main.go`, is in the `golang-samples` repository on GitHub. To get it, retrieve the package of Go samples:

The package retrieval takes a few moments to complete.

Go to the directory of sample code for Profiler in the retrieved package:

The `main.go` program creates a CPU-intensive workload to provide data to the profiler. Start the program and leave it running:

This program is designed to load the CPU as it runs. It is configured to use Profiler, which collects profiling data from the program as it runs and periodically saves it.

After you start the program, you see the `profiler has started` message in a few seconds. In about a minute, two more messages are displayed:

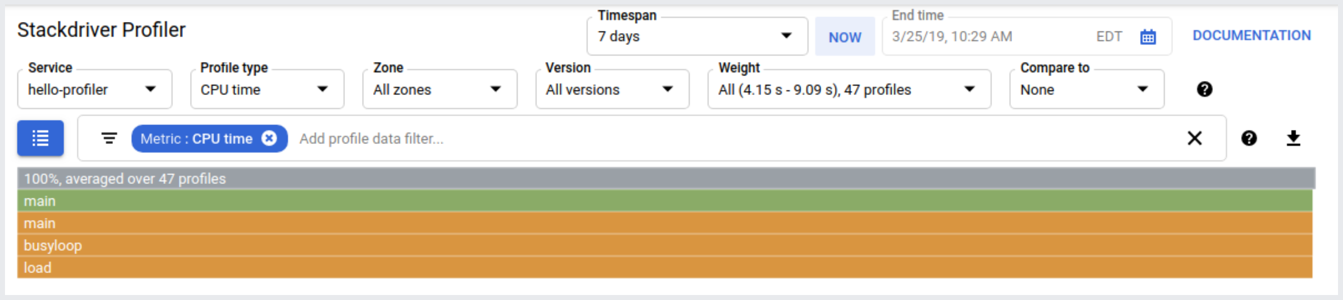
These messages indicate that a profile was created and uploaded to your Cloud Storage project. The program continues to emit the last two messages, about one time per minute, for as long as it runs.

If you receive a permission denied error message after starting the service, see Why am I getting a permission denied error? (/profiler/docs/troubleshooting#permission-denied) for possible causes.

To go to the Profiler interface, select **Profiler** from the Google Cloud Console dashboard:

Go to Profiler (https://console.cloud.google.com/profiler)

The interface offers an array of controls and a flame graph for exploring the profiling data:



Below the time controls are options that let you choose the set of profile data to use. When you are profiling multiple application, you use **Service** to select the origin of the profiled data. **Profile type** lets you choose the kind of profile data to display. **Zone name** and **Version** let you restrict display to data from Compute Engine zones (/compute/docs/regions-zones) or versions of the application. **Weight** lets you select profiles captured during peak resource consumption.

To refine how the flame graph displays the profiles you've selected to analyze, you add filters. In the previous screenshot, the filter bar shows one filter. This filter option is `Metric` and the filter value is `CPU time`.

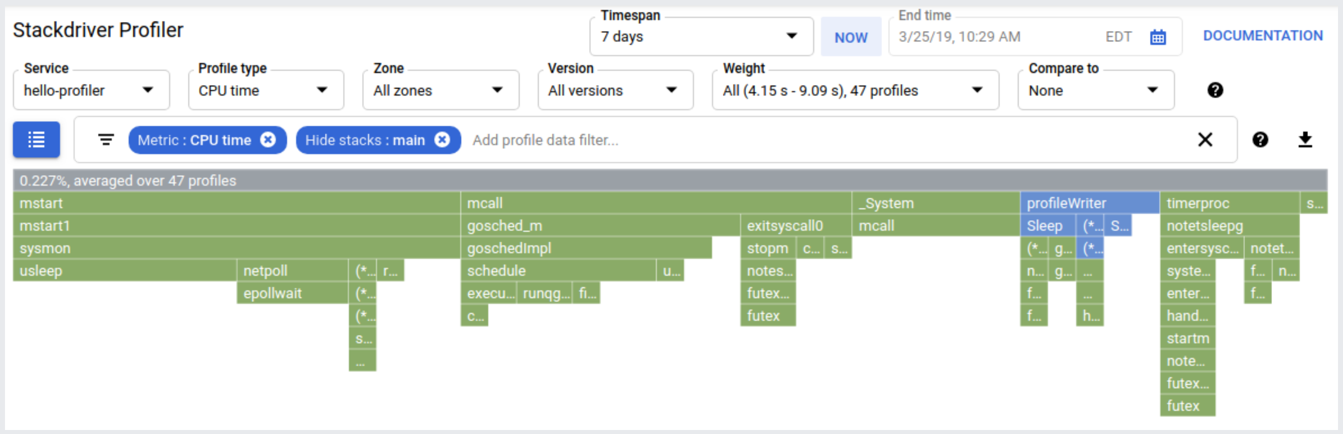
Below the selection controls, the call stacks of the program are displayed in a flame graph. The flame graph represents each function with a frame. The width of the frame represents the proportion of resource consumption by that function. The top frame represents the entire program. This frame always shows 100% of the resource consumption. This frame also lists how many profiles are averaged together in this graph.

The sample program doesn't have a complicated set of call stacks; in the preceding screenshot, you see 5 frames:

- The gray frame represents the entire executable, which accounts for 100% of the resources being consumed.
- The green `main` frame is the Go `runtime.main`.
- The orange `main` frame is the `main` routine of the sample program.
- The orange `busyloop` frame is a routine called from the sample's `main`.
- The orange `main.load` frame is a routine called from the sample's `main`.

The filter selector lets you do things like filter out functions that match some name. For example, if there is a standard library of utility functions, you can remove them from the graph. You can also remove call stacks originating at a certain method or simplify the graph in other ways. The `main.go` application is simple, so there isn't much to filter out.

Even for a simple application, filters let you hide uninteresting frames so that you can more clearly view interesting frames. For example, in the profiling screenshot for the sample code, the gray frame is slightly larger than the first `main` frame under it. Why? Is there something else going on that's not immediately apparent because the `main` call stack consumes such an overwhelming percentage of the resources? To view what is occurring outside of the application's `main` routine, add a filter that hides the call stack of the `main` routine. Only 0.227% of the resource consumption occurs outside of `main`:


Stackdriver Profiler

Timespan: 7 days

End time: 3/25/19, 10:29 AM EDT

Service: hello-profiler
Profile type: CPU time
Zone: All zones
Version: All versions
Weight: All (4.15 s - 9.09 s), 47 profiles
Compare to: None

Metric : CPU time
Hide stacks : main

0.227%, averaged over 47 profiles

See Using the Profiler interface (/profiler/docs/using-profiler) for much more information on filtering and other ways to explore the profiling data.

If the Profiler agent hasn't uploaded any profiles when you start the UI, Profiler displays the message `No data to show`. The message is automatically repla
e Profiler interface after profile data is available.

Need more general information?

- For an overview of Stackdriver Profiler, see About Stackdriver Profiler (/profiler/docs/about-profiler).
- For an introduction to profiling, see Profiling concepts (/profiler/docs/concepts-profiling).
- For detailed information on Profiler features, see Using the Stackdriver Profiler interface (/profiler/docs/using-profiler).

Ready to profile your own application? Choose your language:

- Profiling Go applications (/profiler/docs/profiling-go)
- Profiling Java applications (/profiler/docs/profiling-java)
- Profiling Node.js applications (/profiler/docs/profiling-nodejs)
- Profiling Python applications (/profiler/docs/profiling-python)