GCP Certification Series: 4.3 Managing App Engine resources

Adjusting application traffic splitting parameters.

Prashanta Paudel
Nov 9, 2018 · 8 min read

When you deploy more than one instance of the app having several versions or just many instances with the same version, it is useless until you specify traffic splitting method between them.

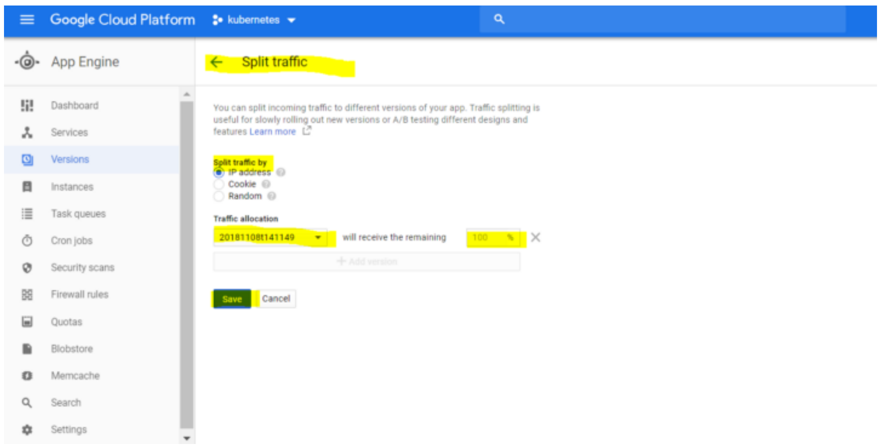
We can split traffic based on

- IP Address
- Cookies

Splitting traffic also helps you deploy rolling update and check for the performance of the new version of the software.

Splitting traffic between same versions

In Console goto Versions inside app engine and click on split traffic then, you will get a prompt where you can specify the type of split and what percentage of traffic to be served by the particular version of the app.



You can split incoming traffic to different versions of your app. Traffic splitting is useful for slowly rolling out new versions or A/B testing different designs and features Learn more

Split traffic by
[✓ IP address] Cookie Random

Traffic allocation
20181108t141149 will receive the remaining 100 %

split traffic

Splitting traffic across multiple versions

When you have specified two or more versions for splitting, you must choose whether to split traffic by using either an IP address or HTTP cookie. It's easier to set up an IP address split, but a cookie split is more precise. For more information, see IP address splitting and Cookie splitting.

IP address splitting

If you choose to split traffic to your application by IP address, when the application receives a request, it hashes the IP address to a value between 0–999 and uses that number to route the request.

IP address splitting has some significant limitations:

- IP addresses are reasonably sticky but are not permanent. Users connecting from cell phones might have a shifting IP address throughout a single session. Similarly, a user on a laptop might be moving from home to a cafe to work, and will also shift through IP addresses. As a result, the user might have an inconsistent experience with your app as their IP address changes.
- Because IP addresses are independently assigned to versions, the resulting traffic split will differ somewhat from what you specify. Although, as your application receives more traffic, the closer the actual split gets to your target. For example, if you ask for 5% of traffic to be delivered to an alternate version, the initial percent of traffic to the version might actually be between 3–7% but eventually averages closer to your target 5%.
- If you need to send internal requests between apps, you should use cookie splitting instead. Requests that are sent between apps running on Google's cloud infrastructure originate from a small number of IP addresses which are likely all assigned to the same version. Therefore, all internal requests might behave similarly to requests sent from a single IP address, meaning that those requests are all routed to the same version. As a result, internal requests do not closely respect the percentages that you set for your IP-based traffic splits. For example, if you set a version to receive 1% of all the traffic to your app and the Google cloud infrastructure addresses were coincidently assigned to that version, then the

actual result might far exceed 1% because all the internal requests are always routed to the assigned version. Requests sent to your app from outside of Google's cloud infrastructure will work as expected since they originate from a varied distribution of IP addresses.

Cookie splitting

If you choose to split traffic to your application by cookies, the application looks in the HTTP request header for a cookie named `GOOGAPPUID`, which contains a value between 0–999:

- If the cookie exists, the value is used to route the request.
- If there is no such cookie, the request is routed randomly.

If the response does not contain the `GOOGAPPUID` cookie, the app first adds the `GOOGAPPUID` cookie with a random value between 0–999 before it is sent.

Using cookies to split traffic makes it easier to accurately assign users to versions. The precision for traffic routing can reach as close as 0.1% to the target split. Although, cookie splitting has the following limitations:

- If you are writing a mobile app or running a desktop client, it needs to manage the `GOOGAPPUID` cookies. For example, when a `Set-Cookie` response header is used, you must store the cookie and include it with each subsequent request. Browser-based apps already manage cookies in this way automatically.
- Splitting internal requests requires extra work. All user requests that are sent from within Google's cloud infrastructure, require that you forward the user's cookie with each request. For example, you must forward the user's cookie in requests sent from your app to another app, or to itself. Note that it is not recommended to send internal requests if those requests don't originate from a user.

Disabling traffic splitting

To disable traffic splitting, you migrate all traffic to a single version.

Traffic splitting via the gcloud CLI

From the `gcloud` CLI, you can use the following subcommand:

```
gcloud app services set-traffic [MY_SERVICE] \
    --splits [MY_VERSION1]=[VERSION1_WEIGHT],[MY_VERSION2]=
[VERSION2_WEIGHT] \
    --split-by [IP_OR_COOKIE]
```

For example:

```
gcloud app services set-traffic \
    --splits 3=.1,2=.9 \
    --split-by cookie
```

Setting Autoscaling Parameters with the API Explorer

Python **2.7**/3.7 |Java 8 |PHP 5.5/7.2 |Go 1.9/1.11 |Node.js

If you use the Cloud SDK tooling to deploy your app, such as `gcloud app deploy`, you can set the following autoscaling parameters in the `app.yaml` configuration file:

- `min_instances`
- `max_instances`
- `target_throughput_utilization`
- `target_cpu_utilization`

However, if you deploy using the `appcfg` tool from the App Engine SDK for Python, you cannot set those autoscaling parameters in the `app.yaml` configuration file. Instead, you must omit these parameters from the configuration file and set them directly in the API explorer user interface, after deploying your app.

To use the API explorer user interface to set an autoscaling parameter:

1. Open the API explorer page.

2. In the right panel under the label *Try This API*, locate the **name** text box and enter the application name string in the following format:

```
apps/<YOUR-PROJECT-ID>/services/default/versions/<YOUR-VERSION-ID>
```

1. Replace `YOUR-PROJECT-ID` with your application's project ID, and `<YOUR-VERSION-ID>` with the version of the app, you are sending the request to. Use the rest of the string as shown.
2. In the **updateMask** text box, enter the full `.json` object path name of the parameter you are setting, using `updateMask` names from the table below:
3. updateMask name `automatic_scaling.standard_scheduler_settings.max_instancesautomatic_scaling.standard_scheduler_settings.min_instances automatic_scaling.standard_scheduler_settings.target_cpu_utilizationautomatic_scaling.standard_scheduler_settings.target_throughput_utilization`
4. If you are setting more than one parameter in one request, supply the mask name for each parameter, separated by a comma. For example, if you are setting the min and max instances, and the CPU utilization, use the following updateMask:

```
automatic_scaling.standard_scheduler_settings.max_instances,
automatic_scaling.standard_scheduler_settings.min_instances,
automatic_scaling.standard_scheduler_settings.target_cpu_utilization
```

1. In the **Request body** box, click **Add request body parameters.**
2. Select **automatic scaling**.
3. Click the hint bubble (the `+` icon), then select **standardSchedulerSettings**.
4. Click the hint bubble, then select the desired auto-scaling scheduler parameter and supply the desired value.
5. To supply another auto-scaling scheduler parameter, click the hint bubble again, select the parameter, and supply its value.

6. The following example shows a sample filled out request body:

```
{
"automaticScaling": {
"standardSchedulerSettings": {
"maxInstances": 100,
"minInstances": 1,
"targetCpuUtilization": 0.75
}
}
}
```

1. Click **Execute**. You may be prompted to authorize API Explorer the first time you run this. If you are prompted, authorize API Explorer by following the prompts.
2. Confirm that the correct settings have been applied by opening the App Engine versions page for your project, and clicking **View** in the *Config* column. You should see the values you just set.

Configuration Files

Python **2.7**/3.7 |Java 8 |PHP 5.5/7.2 |Go 1.9/1.11 |Node.js 8

Note: Services were previously called "modules".Each version of a service is defined in a `.yaml` file, which gives the name of the service and version. The YAML file usually takes the same name as the service it defines, but this is not required. If you are deploying several versions of a service, you can create multiple yaml files in the same directory, one for each version.

Typically, you create a directory for each service, which contains the service's YAML files and associated source code. Optional application-level configuration files (`dispatch.yaml` , `cron.yaml` , `index.yaml` , and `queue.yaml`) are included in the top level app directory. The example below shows three services. In `service1` and `service2` , the source files are at the same level as the YAML file. In `service3` , there are YAML files for two versions.

For small, simple projects, all the app's files can live in one directory:

Every YAML file must include a version parameter. To define the default service, you can explicitly include the parameter `service: default` or leave the service parameter out of the file.

Each service's configuration file defines the scaling type and instance class for a specific service/version. Different scaling parameters are used depending on which type of scaling you specify. If you do not specify scaling, automatic scaling is the default. The scaling and instance class settings are described in the `app.yaml` reference section.

For each service you can also specify settings that map URL requests to specific scripts and identify static files for better server efficiency. These settings are also included in the yaml file and are described in the `app.yaml` reference section.

The default service

Every application has a single default service. You can define the default service in the `app.yaml` with the setting `service: default`, but it isn't necessary to do this. All configuration parameters relevant to services can apply to the default service.

Optional configuration files

These configuration files control optional features that apply to all the services in an app:

- `dispatch.yaml`
- `queue.yaml`
- `index.yaml`
- `cron.yaml`
- `dos.yaml`

To deploy updates of these configuration files to App Engine, run the following command from the directory where they are located:

```
gcloud app deploy [CONFIG_FILE]
```

An example

Here is an example of how you would configure YAML files for an application that has three services: a default service that handles web requests, plus two more services that handle mobile requests and backend processing.

Start by defining a configuration file named `app.yaml` that will handle all web-related requests:

```
runtime: python27
api_version: 1
threadsafe: true
```

If the GCP Console project ID for this app is `simple-sample` then this configuration would create a default service with automatic scaling and a public address of `http://simple-sample.appspot.com` .

Next, assume that you want to create a service to handle mobile web requests. For the sake of the mobile users (in this example) the max pending latency will be just a second and we'll always have at least two instances idle. To configure this you would create a `mobile-frontend.yaml` configuration file. with the following contents:

```
service: mobile-frontend
runtime: python27
api_version: 1
threadsafe: true


automatic_scaling:
  min_idle_instances: 2
  max_pending_latency: 1s
```

The service this file creates would then be reachable at `http://mobile-frontend.simple-sample.appspot.com` .

Finally, add a service, called `my-service` for handling static backend work. This could be a continuous job that exports data from Datastore to BigQuery. The amount of work is relatively fixed, therefore you simply need 1 resident service at any given time. Also, these jobs will need to handle a large amount of in-memory processing, thus you'll want services with an increased memory configuration. To configure this you would create a `my-service.yaml` configuration file with the following contents.

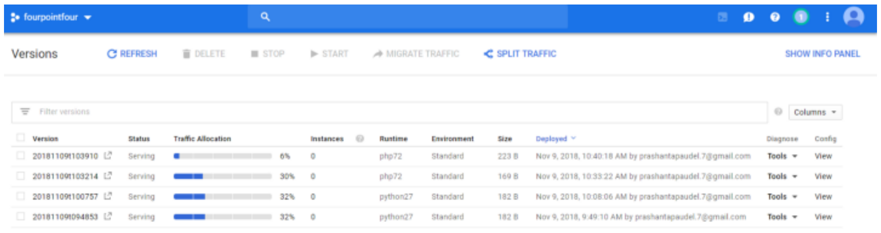
```yaml
service: my-service
runtime: python27
api_version: 1
threadsafe: true


instance_class: B8
manual_scaling:
  instances: 1
```

The service this file creates would then be reachable at `http://my-service.simple-sample.appspot.com`.

Notice the `manual_scaling:` setting. The `instances:` parameter tells App Engine how many instances to create for this service.

You might also want to download this Python demo app and take a look.



Version	Status	Traffic Allocation	Instances	Runtime	Environment	Size	Deployed	Diagnose	Config
20181109t103910	Serving	6%	0	php72	Standard	223 B	Nov 9, 2018, 10:40:18 AM by prashantapaudel.7@gmail.com	Tools	View
20181109t103214	Serving	30%	0	php72	Standard	169 B	Nov 9, 2018, 10:33:22 AM by prashantapaudel.7@gmail.com	Tools	View
20181109t100757	Serving	32%	0	python27	Standard	182 B	Nov 9, 2018, 10:08:06 AM by prashantapaudel.7@gmail.com	Tools	View
20181109t094853	Serving	32%	0	python27	Standard	182 B	Nov 9, 2018, 9:49:10 AM by prashantapaudel.7@gmail.com	Tools	View

Split Traffic