GOOGLE APP ENGINE

CHAPTER 9

PHP: CONFIGURATION

CONFIGURATION

The php.ini File

You can include a php.ini file with your App Engine application. This file lets you customize the behavior of the PHP interpreter directives.

About php.ini

The `php.ini` file should be placed in the base directory of an application (the same directory as the `app.yaml`file). It is loaded when the PHP interpreter is initialized, and before your application code is run.

The file follows the same syntax as other .ini files. A simple example might look like:

```
; This is a simple php.ini file on App Engine

; It enables output buffering for all requests by overriding the

; default setting of the PHP interpreter.

output_buffering = "On"
```

A list of the core directives, along with their changeable mode values, is published on php.net. The `php.ini`directives handled by extensions are documented on the respective pages of the extensions themselves.

You may override any PHP directive that has one of the following changeable mode values:

- `PHP_INI_SYSTEM`
- `PHP_INI_ALL`
- `PHP_INI_PERDIR`

Note that some functions have been disabled in the App Engine implementation of PHP. Directives that target these functions will have no effect.

PHP Directives for App Engine

The following directives are specific to the App Engine environment. They can be included in the php.ini file.

•`google_app_engine.enable_functions` - Functions that have been soft disabled in App Engine, but can be re-enabled using this directive. List the function names in a comma delimited string:

```
google_app_engine.enable_functions = "phpversion, phpinfo"
```

•`google_app_engine.allow_include_gs_buckets` - Allows your application to use the `include` or`require` statements with files stored in Google Cloud Storage. List the buckets containing the files in a comma delimited string:

```
google_app_engine.allow_include_gs_buckets = "bucket_1, bucket_2"
```

You can also specify a bucket and path to files that can be included, for example:

```
google_app_engine.allow_include_gs_buckets = "bucket_1/path_x"
```

When the check is performed for which files may be included from GCS, the supplied path is treated as a prefix that must match the start of the file name in order for it to be included or required. For example using the path example above, the supplied path would allow users to include files from`gs://bucket_1/path_x/...` because the prefix matches but not from `gs://bucket_1 or gs://bucket_1/path_y/` because the prefix doesn't match.

If an uploaded file is moved to an allowed include bucket, a warning is generated to alert the user to a potential < a href="https://en.wikipedia.org/wiki/File_inclusion_vulnerability">LFI attack.. If this happens, you should consider using a more restrictive path.

Note: the path is treated as a file prefix; so it could include a specific file as well, for example,`google_app_engine.allow_include_gs_buckets = "bucket_1/path_x.ext", "bucket_2/path_z/some_file.ext`.

Configuring with app.yaml

A PHP App Engine application can be configured by a file named app.yaml that specifies how URL paths correspond to request handlers and static files. It also contains information about the application code, such as the application ID and the latest version identifier.

Note: If you created your project using the Google Developers Console, your project has a title and an ID. In the instructions that follow, the *project* title and ID can be used wherever an *application* title and ID are mentioned. They are the same thing.

About app.yaml

A PHP app specifies runtime configuration, including versions and URLs, in a file named app.yaml. The following is an example of an app.yaml file for a PHP application:

```
application: myapp

version: 1

runtime: php

api_version: 1

handlers:

# Serve images as static resources.

- url: /(.+\.(gif|png|jpg))$

  static_files: \1

  upload: .+\.(gif|png|jpg)$

  application_readable: true

# Serve php scripts.

- url: /(.+\.php)$

  script: \1
```

The above example will serve files with extension of gif, png, or jpg, i.e. images, as static resources. The files have been configured to be readable by the application code at runtime.

The example will also serve all PHP scripts. If desired, the script handler can be restricted to root-level scripts by using `url: /([^/]+\.php)`. Existing applications may find it useful to simulate Apache mod_rewrite $_GET['q'] routing.

A more extensive `app.yaml` example is provided below:

```yaml
application: myapp

version: 1

runtime: php

api_version: 1

handlers:

- url: /

  script: home.php

- url: /index\.html

  script: home.php

- url: /stylesheets

  static_dir: stylesheets

- url: /(.*\.(gif|png|jpg))$

  static_files: static/\1

  upload: static/.*\.(gif|png|jpg)$

- url: /admin/.*

  script: admin.php

  login: admin

- url: /.*

  script: not_found.php
```

The syntax of `app.yaml` is the YAML format. For more information about this syntax, see the YAML website.

The YAML format supports comments. A line that begins with a pound (#) character is ignored:

```
# This is a comment.
```

URL and file path patterns use POSIX extended regular expression syntax, excluding collating elements and collation classes. Back-references to grouped matches (e.g. `\1`) are supported, as are these Perl extensions: `\w \W \s \S \d \D`

Note: If you factor your application into Modules you will need to specify additional configuration parameters for each module. These are described in the Modules documentation.

Required elements

An `app.yaml` file must include one of each of the following elements:

`application`

The application identifier. This is the identifier you selected when you created the application in the Google Developers Console.

```
application: myapp
```

`version`

A version specifier for the application code. App Engine retains a copy of your application for each version used. An administrator can change which major version of the application is default using the Google Developers Console, and can test non-default versions before making them default. (The default version is loaded when no version is specified or an invalid version is specified.) The version specifier can contain lowercase letters, digits, and hyphens. It cannot begin with the prefix `ah-` and the names `default` and `latest` are reserved and cannot be used. Each version of an application retains its own copy of `app.yaml`. When an application is uploaded, the version mentioned in the `app.yaml` file being uploaded is the version that gets created or replaced by the upload.

```
version: 2-0-test
```

runtime

The name of the App Engine runtime environment used by this application. To specify PHP, use php. Other runtimes are available; please refer to the runtime's documentation for more info. Other JVM languages can customize `app.yaml` based on the specified runtime.

```
runtime: php
```

api_version

The version of the API in the given runtime environment used by this application. When Google releases a new version of a runtime environment's API, your application will continue to use the one for which it was written. To upgrade your application to the new API, you change this value and upload the upgraded code.

At this time, App Engine has one version of the php runtime environment: 1

```
api_version: 1
```

handlers

A list of URL patterns and descriptions of how they should be handled. App Engine can handle URLs by executing application code, or by serving static files uploaded with the code, such as images, CSS or JavaScript. Patterns are evaluated in the order they appear in the `app.yaml`, from top to bottom. The first mapping whose pattern matches the URL is the one used to handle the request.

There are two kinds of handlers: script handlers, and static file handlers. A script handler runs a PHP script in your application to determine the response for the given URL. A static file handler returns the contents of a file, such as an image, as the response. See Script Handlers and Handlers for Static Files below for more information on this value.

```
handlers:

- url: /images

  static_dir: static/images

- url: /.*

  script: myapp.php
```

Notice that you can specify a handler in either of two ways:

- Directly under the `handlers:` element, as shown above.
- Indirectly in `.yaml` files that are included under the `includes:` element.

The following example shows included `.yaml` files, with handlers defined there instead of inside the `app.yaml`file that includes those other `.yaml` files:

```
includes:

- cloud_endpoints.yaml

- web_interface.yaml

- admin_interface.yaml
```

The included `.yaml` files would have the handlers directly defined under `handlers:` element.

Script handlers

A script handler executes a PHP script to handle the request that matches the URL pattern. The mapping defines a URL pattern to match, and the script to be executed.

`url`

The URL pattern, as a regular expression. The expression can contain groupings that can be referred to in the file path to the script with regular expression back-references. For example, `/profile/(.*?)/(.*)` would match the URL `/profile/edit/manager` and use `edit` and `manager` as the first and second groupings.

`script`

Specifies the path to the script from the application root directory:

```
...

handlers:

- url: /profile/(.*?)/(.*)

  script: /employee/\2/\1.php  # specify a script
```

Static file handlers

Static files are files to be served directly to the user for a given URL, such as images, CSS stylesheets, or JavaScript source files. Static file handlers describe which files in the application directory are static files, and which URLs serve them.

For efficiency, App Engine stores and serves static files separately from application files. Static files are not available in the application's file system. If you have data files that need to be read by the application code, the data files must be application files, and must not be matched by a static file pattern.

Static file handlers can be defined in two ways: as a directory structure of static files that maps to a URL path, or as a pattern that maps URLs to specific files.

Static directory handlers

A static directory handler makes it easy to serve the entire contents of a directory as static files. Each file is served using the MIME type that corresponds with its filename extension unless overridden by the directory's`mime_type` setting. All of the files in the given directory are uploaded as static files, and none of them can be run as scripts.

A static directory example:

```
handlers:

# All URLs beginning with /stylesheets are treated as paths to static
files in

# the stylesheets/ directory.

- url: /stylesheets

  static_dir: stylesheets
```

`url`

A URL prefix. This value uses regular expression syntax (and so regexp special characters must be escaped), but it should not contain groupings. All URLs that begin with this prefix are handled by this handler, using the portion of the URL after the prefix as part of the file path.

`static_dir`

The path to the directory containing the static files, from the application root directory. Everything after the end of the matched `url` pattern is appended to `static_dir` to form the full path to the requested file.

All files in this directory are uploaded with the application as static files.

`application_readable`

Optional. By default, files declared in static file handlers are uploaded as static data and are only served to end users, they cannot be read by an application. If this field is set to true, the files are also uploaded as code data so your application can read them. Both uploads are charged against your code and static data storage resource quotas.

`mime_type`

Optional. If specified, all files served by this handler will be served using the specified MIME type. If not specified, the MIME type for a file will be derived from the file's filename extension.

For more information about the possible MIME media types, see the IANA MIME Media Types website.

`http_headers`

Optional. HTTP headers to use for all responses from these URLs.

```
handlers:

- url: /images

  static_dir: static/images

  http_headers:

    X-Foo-Header: foo

    X-Bar-Header: bar value
```

CORS Support

One important use of this feature is to support cross-origin resource sharing (CORS), such as accessing files hosted by another App Engine app.

For example, you could have a game app `mygame.appspot.com` that accesses assets hosted by`myassets.appspot.com`. However, if `mygame` attempts to make a JavaScript `XMLHttpRequest` to `myassets`, it will not succeed unless the handler for `myassets` returns an `Access-Control-Allow-Origin:` response header containing the value `http://mygame.appspot.com`.

Here is how you would make your static file handler return that required response header value:

```
handlers:

- url: /images

  static_dir: static/images

  http_headers:

    Access-Control-Allow-Origin: http://mygame.appspot.com
```

Note: if you wanted to allow everyone to access your assets, you could use the wildcard `*`, instead of`http://mygame.appspot.com`.

`expiration`

Optional. The length of time a static file served by this handler ought to be cached by web proxies and browsers. The value is a string of numbers and units, separated by spaces, where units can be `d` for days, `h` for hours, `m` for minutes, and `s` for seconds. For example, `"4d 5h"` sets cache expiration to 4 days and 5 hours after the file is first requested. See Static cache expiration. If omitted, the application's `default_expiration` is used.

Static file pattern handlers

A static file pattern handler associates a URL pattern with paths to static files uploaded with the application. The URL pattern regular expression can define regular expression groupings to be used in the construction of the file path. You can use this instead of `static_dir` to map to specific files in a directory structure without mapping the entire directory.

Static files cannot be the same as application code files. If a static file path matches a path to a script used in a dynamic handler, the script will not be available to the dynamic handler.

A static file pattern example:

```
handlers:

# All URLs ending in .gif .png or .jpg are treated as paths to static
files in

# the static/ directory. The URL pattern is a regexp, with a grouping that
is

# inserted into the path to the file.

- url: /(.*\.(gif|png|jpg))$

  static_files: static/\1

  upload: static/.*\.(gif|png|jpg)$
```

The following `static_dir` and `static_files` handlers are equivalent:

```
handlers:

- url: /images

  static_dir: static/images

- url: /images/(.*)

  static_files: static/images/\1

  upload: static/images/.*
```

url

The URL pattern, as a regular expression. The expression can contain groupings that can be referred to in the file path to the script with regular expression back-references. For example, `/item-(.*?)/category-(.*)` would match the URL `/item-127/category-fruit`, and use `127` and `fruit` as the first and second groupings.

```
handlers:

- url: /item-(.*?)/category-(.*)

  static_files: archives/\2/items/\1
```

static_files

The path to the static files matched by the URL pattern, from the application root directory. The path can refer to text matched in groupings in the URL pattern.

As in the previous example, `archives/\2/items/\1` inserts the second and first groupings matched in place of`\2` and `\1`, respectively. With the pattern in the example above, the file path would be`archives/fruit/items/127`.

upload

A regular expression that matches the file paths for all files that will be referenced by this handler. This is necessary because the handler cannot determine which files in your application directory correspond with the given `url` and `static_files` patterns. Static files are uploaded and handled separately from application files. The example above might use the following `upload` pattern: `archives/(.*?)/items/(.*)`

application_readable

Optional. By default, files declared in static file handlers are uploaded as static data and are only served to end users, they cannot be read by an application. If this field is set to true, the files are also uploaded as code data so your application can read them. Both uploads are charged against your code and static data storage resource quotas.

mime_type

Optional. If specified, all files served by this handler will be served using the specified MIME type. If not specified, the MIME type for a file will be derived from the file's filename extension. For more information about the possible MIME media types, see the IANA MIME Media Types website.

http_headers

Optional. HTTP headers to use for all responses from these URLs.

```
handlers:

- url: /images/(.*)

  static_files: static/images/\1

  http_headers:
    X-Foo-Header: foo

    X-Bar-Header: bar value
```

`expiration`

Optional. The length of time a static file served by this handler ought to be cached by web proxies and browsers. The value is a string of numbers and units, separated by spaces, where units can be `d` for days, `h` for hours, `m` for minutes, and `s` for seconds. For example, `"4d 5h"` sets cache expiration to 4 days and 5 hours after the file is first requested. See Static cache expiration. If omitted, the application's `default_expiration` is used.

Static cache expiration

Unless told otherwise, web proxies and browsers retain files they load from a website for a limited period of time.

You can define a global default cache period for all static file handlers for an application by including the top-level`default_expiration` element. You can also configure a cache duration for specific static file handlers.

(Script handlers can set cache durations by returning the appropriate HTTP headers to the browser.)

`default_expiration`

Optional. The length of time a static file served by a static file handler ought to be cached by web proxies and browsers, if the handler does not specify its own `expiration`. The value is a string of numbers and units, separated by spaces, where units can be `d` for days, `h` for hours, `m` for minutes, and `s` for seconds. For example, `"4d 5h"` sets cache expiration to 4 days and 5 hours after the file is first requested. If omitted, the production server sets the expiration to 10 minutes.

For example:

```
application: myapp

version: 1

runtime: php

api_version: 1
```

```
default_expiration: "4d 5h"

handlers:

  # ...
```

Important: The expiration time will be sent in the `Cache-Control` and `Expires` HTTP response headers, and therefore, the files are likely to be cached by the user's browser, as well as intermediate caching proxy servers such as Internet Service Providers. Once a file is transmitted with a given expiration time, there is generally **no way** to clear it out of intermediate caches, even if the user clears their own browser cache. Re-deploying a new version of the app will **not** reset any caches. Therefore, if you ever plan to modify a static file, it should have a short (less than one hour) expiration time. In most cases, the default 10-minute expiration time is appropriate.

Secure URLs

Google App Engine supports secure connections via HTTPS for URLs using the `*.appspot.com` domain. When a request accesses a URL using HTTPS, and that URL is configured to use HTTPS in the `app.yaml` file, both the request data and the response data are encrypted by the sender before they are transmitted, and decrypted by the recipient after they are received. Secure connections are useful for protecting customer data, such as contact information, passwords, and private messages.

Note: To use Google Apps domains with HTTPS, you must first activate and configure SSL for App Engine with your domain. Otherwise, users attempting to view pages via https on your domain will likely see timeouts, errors, or warnings.

To configure a URL to accept secure connections, provide a `secure` parameter for the handler:

```
handlers:

- url: /youraccount/.*

  script: accounts.php

  login: required

  secure: always
```

`secure` has the following possible values:

- `optional` - Both HTTP and HTTPS requests with URLs that match the handler succeed without redirects. The application can examine the request to determine which protocol was used, and respond accordingly. This is the default when `secure` is not provided for a handler.

•`never` - Requests for a URL that match this handler that use HTTPS are automatically redirected to the HTTP equivalent URL.

•`always` - Requests for a URL that match this handler that do not use HTTPS are automatically redirected to the HTTPS URL with the same path. Query parameters are preserved for the redirect.

Any URL handler can use the `secure` setting, including script handlers and static file handlers.

When a user's HTTPS query is redirected to be an HTTP query, the query parameters are removed from the request. This prevents a user from accidentally submitting query data over a non-secure connection that was intended for a secure connection.

The development web server does not support HTTPS connections. It ignores the `secure` parameter, so paths intended for use with HTTPS can be tested using regular HTTP connections to the development web server.

To access the versioned appspot.com URL for your application, replace the periods that would usually separate the subdomain components of the URL with the string "`-dot-`". For instance: `https://*desired_version*-dot-*your_app_id*.appspot.com/`.

Google Accounts sign-in and sign-out are always performed using a secure connection, unrelated to how the application's URLs are configured.

Requiring login or administrator status

Any URL handler can have a `login` setting to restrict visitors to only those users who have signed in, or just those users who are administrators for the application. When a URL handler with a `login` setting other than`optional` matches a URL, the handler first checks whether the user has signed in to the application using itsauthentication option. If not, by default, the user is redirected to the sign-in page. You can also use`auth_fail_action` to configure the app to simply reject requests for a handler from users who are not properly authenticated, instead of redirecting the user to the sign-in page.

login

Determines whether the URL handler is restricted to require that the user is signed in. Has three possible values:

- `optional` (the default). Does not require that the user is signed in.
- `required`. If the user has signed in, the handler proceeds normally. Otherwise, the action given in`auth_fail_action` is taken.
- `admin`. As with `required`, performs `auth_fail_action` if the user is not signed in. In addition, if the user is not an administrator for the application, they are given an error message (regardless of the`auth_fail_action` setting). If the user is an administrator, the handler proceeds.

auth_fail_action

Describes the action taken when `login` is present and the user is not logged in. Has two possible values:

- `redirect` (the default). The user is redirected to the Google sign-in page, or `/_ah/login_required` if OpenID authentication is used. The user is redirected back to the application URL after signing in or creating an account.
- `unauthorized`. The request is rejected with an HTTP status code of 401 and an error message.

If an application needs different behavior, the application itself can implement the handling of user logins. Seethe Users API for more information.

The following example requires a login for the `/profile/` directory and an administrator login for the `/admin/`directory:

```
handlers:
- url: /profile/.*
  script: user_profile.php
  login: required
- url: /admin/.*
  script: admin.php
  login: admin
- url: /.*
  script: welcome.php
```

You can configure a handler to refuse access to protected URLs when the user is not signed in, instead of redirecting the user to the sign-in page, by adding auth_fail_action: unauthorized to the handler's configuration:

```
handlers:

- url: /secure_api/.*

  script: api_handler.php

  login: required

  auth_fail_action: unauthorized
```

Skipping files

Files in your application directory whose paths match a static_dir path or a static_files upload path are considered to be static files. All other files in the application directory are considered to be application program and data files.

The skip_files element specifies which files in the application directory are not to be uploaded to App Engine. The value is either a regular expression, or a list of regular expressions. Any filename that matches any of the regular expression is omitted from the list of files to upload when the application is uploaded.

skip_files has the following default:

```
skip_files:

- ^(.*/)?#.*#$

- ^(.*/)?.*~$

- ^(.*/)?.*\.py[co]$

- ^(.*/)?.*/RCS/.*$

- ^(.*/)?\..*$
```

The default pattern excludes Emacs backup files with names of the form #...# and ...~, .pyc and .pyo files, files in an RCS revision control directory, and Unix hidden files with names beginning with a dot (.).

If a new value is specified in app.yaml, it overrides the default value. To extend the above regular express list, copy and paste the above list into your app.yaml and add your own regular expressions. For example, to skip files whose names end in `.bak` in addition to the default patterns, add an entry like this to `skip_files`:

```
skip_files:

- ^(.*/)?\.bak$
```

Defining environment variables

You can define custom environment variables in `app.yaml` using the following format:

```
env_variables:

  foo: 'bar'
```

You can then retrieve the value of these variables by using either `gentev()` or `$_SERVER` (not `$_ENV`). The following commands echo the value of the environment variable `foo`:

```
echo getenv('foo');
```

or

```
echo $_SERVER['foo'];
```

Updated PHP_SELF and SCRIPT_NAME behavior in 1.9.0

The implementation of `$_SERVER['SCRIPT_NAME']` and `$_SERVER['PHP_SELF']` prior to 1.9.0 differs significantly from 1.9.0 and onward. The changes were made to be consistent with the Apache implementation generally expected by PHP applications.

The following examples demonstrate the difference.

Before 1.9.0	After 1.9.0

app.yaml:
`- url: /.*`
`script: index.php`	

`REQUEST_URI: /`
`SCRIPT_FILENAME: /path/to/index.php`
`SCRIPT_NAME: /`
`PHP_SELF: /`
	`REQUEST_URI: /`
	`SCRIPT_FILENAME: /path/to/index.php`
	`SCRIPT_NAME: /index.php`
	`PHP_SELF: /index.php`

`REQUEST_URI: /bar`
`SCRIPT_FILENAME: /path/to/index.php`
`SCRIPT_NAME: /bar`
`PHP_SELF: /bar`
	`REQUEST_URI: /bar`
	`SCRIPT_FILENAME: /path/to/index.php`
	`SCRIPT_NAME: /index.php`
	`PHP_SELF: /index.php`

`REQUEST_URI: /index.php/foo/bar`
`SCRIPT_FILENAME: /path/to/index.php`
`SCRIPT_NAME: /index.php/foo/bar`
`PHP_SELF: /index.php/foo/bar`
	`REQUEST_URI: /index.php/foo/bar`
	`SCRIPT_FILENAME: /path/to/index.php`
	`SCRIPT_NAME: /index.php`
	`PHP_SELF: /index.php/foo/bar`

`REQUEST_URI: /test.php/foo/bar`
`SCRIPT_FILENAME: /path/to/index.php`
`SCRIPT_NAME: /test.php/foo/bar`
`PHP_SELF: /test.php/foo/bar`
	`REQUEST_URI: /test.php/foo/bar`
	`SCRIPT_FILENAME: /path/to/index.php`
	`SCRIPT_NAME: /index.php`
	`PHP_SELF: /index.php`

app.yaml:
`- url: /.*`
`script: foo/index.php`	

`REQUEST_URI: /bar`
`SCRIPT_FILENAME:`
`/path/to/foo/index.php`
`SCRIPT_NAME: /bar`
`PHP_SELF: /bar`
	`REQUEST_URI: /bar`
	`SCRIPT_FILENAME:`
	`/path/to/foo/index.php`
	`SCRIPT_NAME: /foo/index.php`
	`PHP_SELF: /foo/index.php`

Reserved URLs

Several URL paths are reserved by App Engine for features or administrative purposes. Script handler and static file handler paths will never match these paths.

The following URL paths are reserved:

- `/_ah/`
- `/form`

Warmup requests

App Engine frequently needs to load application code into a fresh instance. This happens when you redeploy the application, when the load pattern has increased beyond the capacity of the current instances, or simply due to maintenance or repairs of the underlying infrastructure or physical hardware.

Loading new application code on a fresh instance can result in loading requests. Loading requests can result in increased request latency for your users, but you can avoid this latency using warmup requests. Warmup requests load application code into a new instance before any live requests reach that instance.

App Engine attempts to detect when your application needs a new instance, and (assuming that warmup requests are enabled for your application) initiates a warmup request to initialize the new instance. However, these detection attempts do not work in every case. As a result, you may encounter loading requests, even if warmup requests are enabled in your app. For example, if your app is serving no traffic, the first request to the app will always be a loading request, not a warmup request.

Warmup requests use instance hours like any other request to your App Engine application. In most cases, you won't notice an increase in instance hours, since your application is simply initializing in a warmup request instead of a loading request. Your instance hour usage will likely increase if you decide to do more work (such as precaching) during a warmup request. If you set a minimum number of idle instance, you may encounter warmup requests when those instances first start, but they will remain available after that time.

In PHP, warmup requests are disabled by default. To enable them, add `-warmup` to the `inbound_services`directive in `app.yaml`:

```
inbound_services:

- warmup
```

This causes the App Engine infrastructure to issue `GET` requests to `/_ah/warmup`. You can implement handlers in this directory to perform application-specific tasks, such as pre-caching application data.

Administration console custom pages

If you have administrator-only pages in your application that are used to administer the app, you can have those pages appear in the Administration Console. The Administration Console includes the name of the page in its sidebar, and displays the page in an HTML iframe. To add a page to the Administration Console, add an`admin_console` section in your app's `app.yaml` file, like so:

```
admin_console:

  pages:

  - name: Blog Comment Admin

    url: /blog/admin/comments

  - name: Create a Blog Post

    url: /blog/admin/newentry
```

For each `page`, `url` is the URL path to the page, and `name` is what will appear in the Administration Console navigation. Custom page names can only contain ASCII characters. Custom page urls may be re-routed by rules in the dispatch file, if one exists.

Note: A custom page will appear in the Admin Console only if it is defined in the default version of the default module.

Custom error responses

When certain errors occur, App Engine serves a generic error page. You can configure your app to serve a custom static file instead of these generic error pages, so long as the custom error data is less than 10 kilobytes. You can set up different static files to be served for each supported error code by specifying the files in your app's`app.yaml` file. To serve custom error pages, add a `error_handlers` section to your `app.yaml`, as in this example:

```
error_handlers:

  - file: default_error.html

  - error_code: over_quota

    file: over_quota.html
```

Warning: Make sure that the path to the error response file does not overlap with static file handler paths.

Each `file` entry indicates a static file that should be served in place of the generic error response. The`error_code` indicates which error code should cause the associated file to be served. Supported error codes are as follows:

- `over_quota`, which indicates the app has exceeded a resource quota;
- `dos_api_denial`, which is served to any client blocked by your app's DoS Protection configuration;
- `timeout`, served if a deadline is reached before there is a response from your app.

The `error_code` is optional; if it's not specified, the given file is the default error response for your app.

You can optionally specify a `mime_type` to use when serving the custom error. Seehttp://www.iana.org/assignments/media-types/ for a complete list of MIME types.

Custom PageSpeed configuration

Beta

App Engine's support for PageSpeed is an experimental, innovative, and rapidly changing new feature for Google App Engine. Unfortunately, being on the bleeding edge means that we may make backwards-incompatible changes to App Engine's support for PageSpeed. We will inform the community when this feature is no longer in Beta.

When enabled, PageSpeed Service will be applied globally to your application. All versions of your app will be automatically optimized with the same configuration, that of the most-recently updated version. If you want to try a new configuration, perhaps to test some "risky" optimizations, you might expect that you could do so in a test Application Version while your users continue to use the default Application Version. But since updating the test version's configuration applies to all versions, the "risky" settings are applied to the version that your users use, too. Instead, to try out these settings, you could

- Turn off PageSpeed for your application, update PageSpeed configuration, and view your site using the PageSpeed chrome extension; or
- Copy the relevant parts of your application to a separate test application that has its own application ID and versions.

PageSpeed is a family of tools for optimizing the performance of web pages. You can use the Application Console to enable PageSpeed with a solid set of safe default optimizations. You can also edit your application configuration to fine-tune PageSpeed. You can configure PageSpeed to ignore some URLs; you can turn on some "risky" optimizations that don't work for all sites but might work for yours. For a custom PageSpeed configuration, you can add a `pagespeed` section to your application configuration. An example that shows the possible parts (but not all choices) of `pagespeed`:

```
pagespeed:

  domains_to_rewrite:

  - www.foo.com

  - https://*.secure.foo.com

  url_blacklist:

  - http://myapp.com/blob/*

  - http://myapp.com/comments/*

  enabled_rewriters:

  - MinifyCss

  disabled_rewriters:

  - ImageStripColorProfile
```

This section may have any or all of the following parts:

`domains_to_rewrite`
: Which domains' content PageSpeed should rewrite. Normally, PageSpeed only rewrites data served by your application. But you can tell PageSpeed to rewrite content from other domains when your application shows that data. For example, your application might use a free image hosting service on some other domain. Since the image hosting site isn't part of your application, PageSpeed doesn't optimize those images by default. You *might* want it to optimize these images: they can be compressed and cached by Google, and thus displayed faster. You might *not* want it to optimize these images: Since pagespeed optimizes the images, you will be charged whenever those images are served.

`url_blacklist`
: This is a list of URLS; the `*` character is a wildcard. PageSpeed won't try to optimize URLs matching these URLs. This can be especially useful if you turn on some "risky" optimizations that break content for some URLs but make the rest of your site much much faster. By default, there is no blacklist.

`enabled_rewriters`, `disabled_rewriters`

By default, some "safe" optimization rewriters are turned on. Other "risky" optimizations are not turned on. You can choose more optimizations to use by listing them in `enabled_rewriters`; you can turn off default rewriters by listing them in `disabled_rewriters`. The following rewriters are available:

HTML Rewriters

- ProxyCss - *default*
- ProxyImages - *default*
- ProxyJs - *default*
- ConvertMetaTags - *default*
- InlineCss - *default*
- InlineJs
- InlineImages
- InlinePreviewImages - *default*
- CollapseWhitespace
- CombineHeads
- ElideAttributes
- RemoveComments
- RemoveQuotes
- LeftTrimUrls

CSS Rewriters

- CombineCss - *default*
- MoveCssToHead - *default*
- MinifyCss

Image Rewriters

- WebpOptimization - *default*
- ImageConvertToJpeg - *default*
- ImageRecompressJpeg - *default*
- ImageProgressiveJpeg - *default*
- ImageRecompressPng - *default*
- ImageStripMetadata - *default*
- ImageStripColorProfile - *default*
- ImageResize - *default*
- LazyloadImages - *default*
- ImageAddDimensions

JavaScript Rewriters

- CombineJs - *default*
- JsOptimize - *default*
- DeferJs

Simulate Apache mod_rewrite routing

Basic Apache `mod_rewrite` functionality can be simulated through the use of a PHP script referenced from`app.yaml` that will in turn load the desired script. This example simulates the common PHP pattern that expects the variable `$_GET['q']` to contain the request path.

About `mod_rewrite.php`

To enable mod_rewrite functionality, your application needs to include `mod_rewrite.php`, which is the script that will be invoked for all requests to your application to perform the request routing. As described in the comments the script will check for the existance of root-level PHP scripts and invoke them while placing the path portion of `$_SERVER['REQUEST_URI']` in the `$_GET['q']` variable.

```php
<?php
/**
 * @file
 * Provide basic mod_rewrite like functionality.
 * Pass through requests for root php files and forward all other requests
to index.php with $_GET['q'] equal to path. The following are examples
that demonstrate how a request using mod_rewrite.php will appear to a PHP
script.
 *
 * - /install.php: install.php
 * - /update.php?op=info: update.php?op=info
 * - /foo/bar: index.php?q=/foo/bar
 * - /: index.php?q=/
 */
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
// Provide mod_rewrite like functionality. If a php file in the root
directory
// is explicitly requested then load the file, otherwise load index.php
and
// set get variable 'q' to $_SERVER['REQUEST_URI'].
if (dirname($path) == '/' && pathinfo($path, PATHINFO_EXTENSION) == 'php')
{
  $file = pathinfo($path, PATHINFO_BASENAME);
}
```

```
else {
  $file = 'index.php';
  // Provide mod_rewrite like functionality by using the path which
excludes
  // any other part of the request query (ie. ignores ?foo=bar).
  $_GET['q'] = $path;
}
// Override the script name to simulate the behavior without
mod_rewrite.php.
// Ensure that $_SERVER['SCRIPT_NAME'] always begins with a / to be
consistent
// with HTTP request and the value that is normally provided.
$_SERVER['SCRIPT_NAME'] = '/' . $file;
require $file;
```

Example app

The following show a very simple application written to expect `$_GET['q']`.

`app.yaml`

As you can see from this `app.yaml` file, this application will provide two root-level PHP scripts and serve static files out of a directory named `downloads`.

```
application: mod_rewrite_simulator
version: 1
runtime: php
api_version: 1
handlers:
# Example of handler which should be placed above the catch-all handler.
- url: /downloads
  static_dir: downloads
# Catch all unhandled requests and pass to mod_rewrite.php which will
simulate
# mod_rewrite by forwarding the requests to index.php?q=... (or other
root-level
# PHP file if specified in incoming URL.
- url: /.*
  script: mod_rewrite.php
```

`index.php`

`index.php` is a router-style `index.php` which reads `$_GET['q']` to determine the request path.

```
<?php

if ($_GET['q'] == '/help') {

  echo 'This is some help text.';

  exit;

}

echo 'Welcome to the site!';
```

`other.php`

The following is an example of a root-level script that can be called directly. Many PHP frameworks have scripts like `install.php` or `update.php` that would behave similarly.

```
<?php

echo 'Welcome to the other site.';
```

Example requests

Given the above example application the following requests would be handled as shown.

- `/` translates to `index.php` with `$_GET['q'] = '/'`
- `/help` translates to `index.php` with `$_GET['q'] = '/help'`
- `/other.php` translates to `other.php` with `$_GET['q'] = null`
- `/downloads/foo_17.png` translates to `downloads/foo_17.png`

Avoid the need for `mod_rewrite.php`

Many PHP frameworks no longer depend on `$_GET['q']`. Instead they make use of`$_SERVER['REQUEST_URI']` which works both with and without `mod_rewrite`. As such the latter is the preferred method on App Engine.

As used in `mod_rewrite.php` a clean method of utilizing `REQUEST_URI` is the following:

```
<?php

$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

if ($path == '/help') {

  echo 'This is some help text.';

  exit;

}

echo 'Welcome to the site!';
```

Scheduled Tasks With Cron for PHP

The App Engine Cron Service allows you to configure regularly scheduled tasks that operate at defined times or regular intervals. These tasks are commonly known as *cron jobs*. These cron jobs are automatically triggered by the App Engine Cron Service. For instance, you might use this to send out a report email on a daily basis, to update some cached data every 10 minutes, or to update some summary information once an hour.

A cron job will invoke a URL, using an HTTP GET request, at a given time of day. An HTTP request invoked by cron can run for up to 10 minutes, but is subject to the same limits as other HTTP requests.

Free applications can have up to 20 scheduled tasks. Paid applications can have up to 100 scheduled tasks.

About cron.yaml

A `cron.yaml` file in the root directory of your application (alongside `app.yaml`) configures scheduled tasks for your PHP application. The following is an example `cron.yaml` file:

```
cron:

- description: daily summary job

  url: /tasks/summary

  schedule: every 24 hours

- description: monday morning mailout

  url: /mail/weekly

  schedule: every monday 09:00

  timezone: Australia/NSW

- description: new daily summary job

  url: /tasks/summary

  schedule: every 24 hours

  target: beta
```

The syntax of `cron.yaml` is the YAML format. For more information about this syntax, see the YAML website for more information.

A `cron.yaml` file consists of a number of job definitions. A job definition must have a `url` and a `schedule`. You can also optionally specify a `description`, `timezone`, and a `target`. The description is visible in the Admin Console and the development server's admin interface.

The `url` field specifies a URL in your application that will be invoked by the Cron Service. See Securing URLs for Cron for more. The format of the schedule field is covered in The Schedule Format.

The `timezone` should be the name of a standard zoneinfo time zone name. If you don't specify a timezone, the schedule will be in UTC (also known as GMT).

The `target` string is prepended to your app's hostname. It is usually the name of a module. The cron job will be routed to the default version of the named module. Note that if the default version of the module changes, the job will run in the new default version.

Warning: Be careful if you run a cron job with traffic splitting enabled. The request from the cron job is always sent from the same IP address, so if you've specified IP address splitting, the logic will route the request to the same version every time. If you've specified cookie splitting, the request will not be split at all, since there is no cookie accompanying the request.

If there is no module with the name assigend to `target`, the name is assumed to be an app version, and App Engine will attempt to route the job to that version. See PHP Application Configuration for more information about application versions.

If you use a dispatch file, your job may be re-routed. For example, given the following cron.yaml and dispatch.yaml files, the job will run in module2, even though its target is module1:

```
cron:

- description: test dispatch vs target

 url: /tasks/hello_module2

 schedule: every 1 mins

 target: module1

dispatch.yaml:

dispatch:

- url: '*/tasks/hello_module2'

 module: module2
```

The schedule format

Cron schedules are specified using a simple English-like format.

The following are examples of schedules:

```
every 12 hours

every 5 minutes from 10:00 to 14:00

2nd,third mon,wed,thu of march 17:00

every monday 09:00

1st monday of sep,oct,nov 17:00

every day 00:00
```

If you don't need to run a recurring job at a specific time, but instead only need to run it at regular intervals, use the form:

```
every N (hours|mins|minutes) ["from" (time) "to" (time)]
```

The brackets are for illustration only, and quotes indicate a literal.

- *N* specifies a number.
- *hours* or *minutes* (you can also use *mins*) specifies the unit of time.
- *time* specifies a time of day, as HH:MM in 24 hour time.

By default, an interval schedule starts the next interval after the last job has completed. If a *from...to* clause is specified, however, the jobs are scheduled at regular intervals independent of when the last job completed. For example:

```
every 2 hours from 10:00 to 14:00
```

This schedule runs the job three times per day at 10:00, 12:00, and 14:00, regardless of how long it takes to complete. You can use the literal "synchronized" as a synonym for *from 00:00 to 23:59*:

```
every 2 hours synchronized
```

If you want more specific timing, you can specify the schedule as:

```
("every"|ordinal) (days) ["of" (monthspec)] (time)
```

Where:

- *ordinal* specifies a comma separated list of "1st", "first" and so forth (both forms are ok)
- *days* specifies a comma separated list of days of the week (for example, "mon", "tuesday", with both short and long forms being accepted); "every day" is equivalent to "every mon,tue,wed,thu,fri,sat,sun"
- *monthspec* specifies a comma separated list of month names (for example, "jan", "march", "sep"). If omitted, implies every month. You can also say "month" to mean every month, as in "1,8,15,22 of month 09:00".
- *time* specifies the time of day, as HH:MM in 24 hour time.

Securing URLs for cron

A cron handler is just a normal handler defined in `app.yaml`. You can prevent users from accessing URLs used by scheduled tasks by restricting access to administrator accounts. Scheduled tasks can access admin-only URLs. You can restrict a URL by adding `login: admin` to the handler configuration in `app.yaml`.

An example might look like this in `app.yaml`:

```
application: hello-cron

version: 1

runtime: php

api_version:

handlers:

- url: /report/weekly

  script:

  login: admin
```

Note: While cron jobs can use URL paths restricted with `login: admin`, they *cannot* use URL paths restricted with `login: required`.

For more information see PHP Application Configuration: Requiring Login or Administrator Status.

To test a cron job, sign in as an administrator and visit the URL of the handler in your browser.

Requests from the Cron Service will also contain a HTTP header:

```
X-AppEngine-Cron: true
```

The `X-AppEngine-Cron` header is set internally by Google App Engine. If your request handler finds this header it can trust that the request is a cron request. If the header is present in an external user request to your app, it is stripped, except for requests from logged in administrators of the application, who are allowed to set the header for testing purposes.

Google App Engine issues Cron requests from the IP address `0.1.0.1`.

Cron and app versions

If the `target` parameter has been set for a job, the request is sent to the specified version. Otherwise Cron requests are sent to the default version of the application.

Uploading cron jobs

You can use `appcfg.py` to upload cron jobs and view information about the defined cron jobs. When you upload your application to App Engine using `appcfg.py update`, the Cron Service is updated with the contents of`cron.yaml`. You can update just the cron configuration without uploading the rest of the application using`appcfg.py update_cron`.

To delete all cron jobs, change the `cron.yaml` file to just contain:

```
cron:
```

You can display the parsed version of your cron jobs, including the times the jobs will run, using the `appcfg.py cron_info` command.

Note that `appcfg.py cron_info` will not correctly compute schedules if a timezone different than UTC is specified.

Cron support in the Admin Console

The Admin Console allows you to view the state of your cron jobs. Select the "Cron Jobs" link from the side menu to view the state of the jobs, including the last time the job was run and the result of the job.

You can also see when cron jobs are added or removed by selecting the "Admin Logs" page from the Admin Console menu.

Cron support in the development server

When using the PHP SDK, the dev_appserver has an admin interface that allows you to view cron jobs at`http://localhost:8000/cron`.

The development server doesn't automatically run your cron jobs. You can use your local desktop's cron or scheduled tasks interface to trigger the URLs of your jobs with curl or a similar tool.

PHP Task Queue Configuration

Applications define task queues in a configuration file called queue.yaml. You can use queue.yaml to configure push queues . This configuration file is optional for push queues, which have a default queue.

About queues

An app can define task queues using a file named queue.yaml. The file specifies a directive named queue. Within this directive, you can name any number of individual queues and define their processing rates.

The app's queue configuration applies to all versions of the app. If a given version of an app enqueues a task, the queue uses the task handler for that version of the app. To control this behavior, see the target parameter.

An app can only add tasks to queues defined in queue.yaml and the default queue. If you upload a newqueue.yaml file that removes a queue, but that queue still has tasks, the queue is "paused" (its rate is set to 0) but not deleted. You can re-enable the deleted queue by uploading a new queue.yaml file with the queue defined.

Setting the storage limit for all queues

You can use queue.yaml to define the total amount of storage that task data can consume over all queues. To define the total storage limit, include a directive named total_storage_limit at the top level just above eachqueue line, like this:

```
# Set the total storage limit for all queues to 120MB

total_storage_limit: 120M

queue:

- name: foo

  rate: 35/s
```

The value is a number followed by a unit: B for bytes, K for kilobytes, M for megabytes, G for gigabytes, T for terabytes. For example, 100K specifies a limit of 100 kilobytes. If adding a task would cause the queue to exceed its storage limit, the call to add the task will fail. The default limit is 500M (500 megabytes) for free apps. For billed apps there is no limit until you explicitly set one. You can use this limit to protect your app from a fork bomb programming error in which each task adds multiple other tasks during its execution. If your app is receiving errors for insufficient quota when adding tasks, increasing the total storage limit may help. If you are using this feature, we strongly recommend setting a limit that corresponds to the storage required for several days' worth of tasks. In this way, your app is robust to its queues being temporarily backed up and can continue to accept new tasks while working through the backlog while still being protected from a fork bomb programming error.

Defining push queues and processing rates

You can define any number of individual queues by providing a queue `name`. You can control the rate at which tasks are processed in each queue by defining other directives, such as `rate`, `bucket_size`, and`max_concurrent_requests`.

You can read more about these directives in the Queue Definitions section.

The task queue uses token buckets to control the rate of task execution. Each named queue has a token bucket that holds a certain number of tokens, defined by the `bucket_size` directive. Each time your application executes a task, it uses a token. Your app continues processing tasks in the queue until the queue's bucket runs out of tokens. App Engine refills the bucket with new tokens continuously based on the `rate` that you specified for the queue.

Configuring the default queue

All apps have a push queue named `default`. This queue has a preset rate of 5 tasks per second, but you can change this rate by defining a default queue in `queue.yaml`. If you do not configure a default queue in`queue.yaml`, the `default` queue doesn't display in the Administration Console until the first time it is used. You can customize the settings for this queue by defining a queue named `default` in `queue.yaml`:

```
queue:

# Change the refresh rate of the default queue from 5/s to 1/s

- name: default

  rate: 1/s
```

Configuring the processing rate

If your queue contains tasks to process, and the queue's bucket contains tokens, App Engine processes as many tasks as there are tokens remaining in the bucket. This can lead to bursts of processing, consuming system resources and competing with user-serving requests.

If you want to prevent too many tasks from running at once or to prevent datastore contention, you use`max_concurrent_requests`.

The following samples shows how to set `max_concurrent_requests` to limit tasks and also shows how to adjust the bucket size and rate based on your application's needs and available resources:

```
queue:

- name: optimize-queue

  rate: 20/s

  bucket_size: 40

  max_concurrent_requests: 10
```

Configuring retry attempts for failed tasks

Tasks executing in the task queue can fail for many reasons. If a task fails to execute (by returning any HTTP status code outside of the range 200–299), App Engine retries the task until it succeeds. By default, the system gradually reduces the retry rate to avoid flooding your application with too many requests, but schedules retry attempts to recur at a maximum of once per hour until the task succeeds.

Retrying tasks in push queues

In push queues, you can specify your own scheme for task retries by adding the retry_parameters directive in`queue.yaml`. This addition allows you to specify the maximum number of times to retry failed tasks in a specific queue. You can also set a time limit for retry attempts and control the interval between attempts.

The following example demonstrates various retry scenarios:

- In `fooqueue`, tasks are retried at least seven times and for up to two days from the first execution attempt. After both limits are passed, it fails permanently.

- In barqueue, App Engine attempts to retry tasks, increasing the interval linearly between each retry until reaching the maximum backoff and retrying indefinitely at the maximum interval (so the intervals between requests are 10s, 20s, 30s, ..., 190s, 200s, 200s, ...).
- In bazqueue, the interval increases to twice the minimum backoff and retries indefinitely at the maximum interval (so the intervals between requests are 10s, 20s, 40s, 80s, 120s, 160s, 200s, 200s, ...).

```
queue:
- name: fooqueue
  rate: 1/s
  retry_parameters:
    task_retry_limit: 7
    task_age_limit: 2d
- name: barqueue
  rate: 1/s
  retry_parameters:
    min_backoff_seconds: 10
    max_backoff_seconds: 200
    max_doublings: 0
- name: bazqueue
  rate: 1/s
  retry_parameters:
    min_backoff_seconds: 10
    max_backoff_seconds: 200
    max_doublings: 3
```

Queue definitions

The `queue.yaml` file is a YAML file whose root directive is `queue-entries`. This directive contains zero or more`queue` directives, one for each named queue.

A `queue` directive contains configuration for a queue. It can contain the following directives:

`bucket_size` (push queues only)

Task queues use a "token bucket" algorithm for dequeueing tasks. The bucket size limits how fast the queue is processed when many tasks are in the queue and the rate is high. *The maximum value for bucket size is 100.* This allows you to have a high rate so processing starts shortly after a task is enqueued, but still limit resource usage when many tasks are enqueued in a short period of time.

If no `bucket_size` is specified for a queue, the default value is 5.

For more information on the algorithm, see the Wikipedia article on token buckets.

`max_concurrent_requests` (push queues only)

Sets the maximum number of tasks that can be executed at any given time in the specified queue. The value is an integer. By default, this directive is unset and there is no limit on the maximum number of concurrent tasks. One use of this directive is to prevent too many tasks from running at once or to prevent datastore contention.

Restricting the maximum number of concurrent tasks gives you more control over your queue's rate of execution. For example, you can constrain the number of instances that are running the queue's tasks. Limiting the number of concurrent requests in a given queue allows you to make resources available for other queues or online processing.

`name`

The name of the queue.

A queue name can contain letters, numbers, and hyphens.

`rate` **(push queues only)**

How often tasks are processed on this queue. The value is a number followed by a slash and a unit of time, where the unit is `s` for seconds, `m` for minutes, `h` for hours, or `d` for days. For example, the value `5/m` says tasks will be processed at a rate of 5 times per minute.

If the number is `0` (such as `0/s`), the queue is considered "paused," and no tasks are processed.

`retry_parameters`

Configures retry attempts for failed tasks. This addition allows you to specify the maximum number of times to retry failed tasks in a specific queue. You can also set a time limit for retry attempts and control the interval between attempts.

`task_retry_limit`
: The maximum number of retry attempts for a failed task. If specified with `task_age_limit`, App Engine retries the task until both limits are reached.

`task_age_limit` **(push queues only)**
: The time limit for retrying a failed task, measured from when the task was first run. The value is a number followed by a unit of time, where the unit is `s` for seconds, `m` for minutes, `h` for hours, or `d` for days. For example, the value `5d` specifies a limit of five days after the task's first execution attempt. If specified with `task_retry_limit`, App Engine retries the task until both limits are reached.

`min_backoff_seconds` **(push queues only)**
: The minimum number of seconds to wait before retrying a task after it fails.

`max_backoff_seconds` **(push queues only)**
: The maximum number of seconds to wait before retrying a task after it fails.

`max_doublings` **(push queues only)**
: The maximum number of times that the interval between failed task retries will be doubled before the increase becomes constant. The constant is: 2**(max_doublings - 1) * min_backoff_seconds.

`target` **(push queues only)**

A string naming a module/version, a frontend version, or a backend, on which to execute all of the tasks enqueued onto this queue.

The string is prepended to the domain name of your app when constructing the HTTP request for a task. For example, if your app ID is `my-app` and you set the target to `my-version.my-module`, the URL hostname will be set to `my-version.my-module.my-`

app.appspot.com.

If target is unspecified, then tasks are invoked on the same version of the application where they were enqueued. So, if you enqueued a task from the default application version without specifying a target on the queue, the task is invoked in the default application version. Note that if the default application version changes between the time that the task is enqueued and the time that it executes, then the task will run in the new default version.

Note: If you are using modules along with a dispatch file, your task's HTTP request may be intercepted and re-routed to another module.

Updating task queue configuration

The task queue configuration for the app is updated when you upload the application using `appcfg.py update`. You can update just the configuration for an app's task queues without uploading the full application. To upload the `queue.yaml` file, use the `appcfg.py update_queues` command:

```
appcfg.py update_queues myapp/
```

See Uploading and Managing a PHP App for more information.

DoS Protection Service for PHP

The App Engine Denial of Service (DoS) Protection Service enables you to protect your application from running out of quota when subjected to denial of service attacks or similar forms of abuse. You can blacklist IP addresses or subnets, and requests routed from those addresses or subnets will be dropped before your application code is called. No resource allocations, billed or otherwise, are consumed for these requests.

Do not use this service for security. It is designed for quantitative abuse prevention, such as preventing DoS attacks, only. Some requests from blacklisted users may still get through to your application.

By default, App Engine serves a generic error page to blacklisted addresses. You can configure your app to serve a custom response instead.

About dos.yaml

A `dos.yaml` file in the root directory of your application (alongside `app.yaml`) configures DoS Protection Service blacklists for your application. The following is an example `dos.yaml` file:

```
blacklist:

- subnet: 1.2.3.4

  description: a single IP address

- subnet: 1.2.3.4/24

  description: an IPv4 subnet

- subnet: abcd::123:4567

  description: an IPv6 address

- subnet: abcd::123:4567/48

  description: an IPv6 subnet
```

The syntax of `dos.yaml` is the YAML format. For more information about this syntax, see the YAML website.

A `dos.yaml` file consists of a number of blacklist entries. A blacklist entry has a `subnet`, and can optionally specify a `description`. The description will be visible in the Admin Console. The `subnet` is any valid IPv4 or IPv6 subnet in CIDR notation.

Limits

You may define a maximum of 100 blacklist entries in your configuration file. Uploading a configuration file with more than 100 entries will fail.

Uploading DoS configuration

You can use `appcfg.py` to upload DoS configs. When you upload your application to App Engine using`appcfg.py update`, the DoS Protection Service is updated with the contents of `dos.yaml`. The new config will be viewable using the Administration Console straight away, but may take a few minutes to take effect. You can update just the DoS configuration without uploading the rest of the application using `appcfg.py update_dos`.

To delete all blacklist entries, change the `dos.yaml` file to just contain:

```
blacklist:
```

Viewing uploaded DoS configuration in the Administration Console

The Administration Console allows you to view your current DoS configuration, if you have uploaded one. Select the "Blacklist" link from the side menu, and you should see a table of blacklisted subnets/IP addresses and their associated descriptions from your configuration file.

Viewing top users in the Administration Console

The Administration Console also allows you to view the top users that have recently been hitting your site, even if you haven't uploaded a DoS configuration. Select the "Blacklist" link from the side menu, and you should see a table of the top IP addresses that have been hitting your application and how many requests have been recorded. It will be below the table of blacklists, if any. IP addresses that you have blacklisted will not show up in this table.