GOOGLE APP ENGINE

CHAPTER 6

PHP: MODULES

MODULES

App Engine Modules in PHP

App Engine Modules (or just "Modules" hereafter) is a feature that lets developers factor large applications into logical components that can share stateful services and communicate in a secure fashion. An app that handles customer requests might include separate modules to handle other tasks, such as:

- API requests from mobile devices
- Internal, admin-like requests
- Backend processing such as billing pipelines and data analysis

Modules can be configured to use different runtimes and to operate with different performance settings.

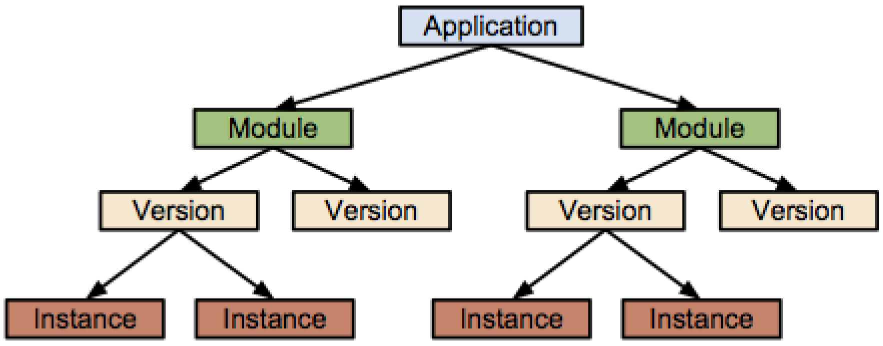
Application hierarchy

At the highest level, an App Engine application is made up of one or more *modules*. Each module consists of source code and configuration files. The files used by a module represent a *version* of the module. When you deploy a module, you always deploy a specific version of the module. For this reason, whenever we speak of a module, it usually means a version of a module.

You can deploy multiple versions of the same module, to account for alternative implementations or progressive upgrades as time goes on.

Every module and version must have a name. A name can contain numbers, letters, and hyphens. It cannot be longer than 63 characters and cannot start or end with a hyphen.

While running, a particular module/version will have one or more *instances*. Each instance runs its own separate executable. The number of instances running at any time depends on the module's scaling type and the amount of incoming requests:



Stateful services (such as Memcache and Task Queues) are shared by all the modules in an application. Every module, version, and instance has its own unique URI (for example, `v1.my-module.my-app.appspot.com`). Incoming user requests are routed to an instance of a particular module/version according to URL addressing conventions and an optional customized dispatch file.

Please note that in April of 2013, Google stopped issuing SSL certificates for double-wildcard domains hosted at `appspot.com` (i.e. `*.*.appspot.com`). If you rely on such URLs for HTTPS access to your application, please change any application logic to use "-dot-" instead of ".". For example, to access version "1" of application "myapp" use "https://1-dot-myapp.appspot.com" instead of "https://1.myapp.appspot.com." If you continue to use "https://1.myapp.appspot.com" the certificate will not match, which will result in an error for any User-Agent that expects the URL and certificate to match exactly.

Instance scaling and class

While an application is running, incoming requests are routed to an existing or new instance of the appropriate module/version. The scaling type of a module/version controls how instances are created. There are three scaling types: *manual*, *basic*, and *automatic*.

Manual Scaling

A module with manual scaling runs continuously, allowing you to perform complex initialization and rely on the state of its memory over time.

Basic Scaling

A module with basic scaling will create an instance when the application receives a request. The instance will be turned down when the app becomes idle. Basic scaling is ideal for work that is intermittent or driven by user activity.

Automatic Scaling

Automatic scaling is the scaling policy that App Engine has used since its inception. It is based on request rate, response latencies, and other application metrics. Previously users could use the Admin Console to configure the automatic scaling parameters (instance class, idle instances and pending latency) for an application's frontend versions only. These settings now apply to every version of every module that has automatic scaling.

Each scaling type offers a selection of instance classes, with different amounts of CPU and Memory. The following tables list the features of the three types of scaling, and the service levels and costs of the various instance classes:

Scaling Types

Feature	Automatic Scaling	Manual Scaling	Basic Scaling
Deadlines	60-second deadline for HTTP requests, 10-minute deadline for tasks	Requests can run indefinitely. A manually-scaled instance can choose to handle \`/_ah/start\` and execute a program or script for many hours without returning an HTTP response code. Tasks can run up to 24 hours.	Same as manual scaling.
CPU/Memory	Configurable by selecting an F1, F2, F4, or F4_1G instance class	Configurable by selecting a B1, B2, B4, B4_1G, or B8 instance class	Configurable by selecting a B1, B2, B4, B4_1G, or B8 instance class
Residence	Instances are evicted from memory based on usage patterns.	Instances remain in memory, and state is preserved across requests. When instances are restarted, an \`/_ah/stop\` request appears in the logs. If there is a registered stop callback method, it has 30 seconds to complete before shutdown occurs.	Instances are evicted based on the \`idle_timeout\` parameter. If an instance has been idle, i.e. has not received a request, for more than \`idle_timeout\`, then the instance is evicted.
Startup and Shutdown	Instances are created on demand to handle requests and automatically turned down when idle.	Instances are sent a start request automatically by App Engine in the form of an empty GET request to \`/_ah/start\`. An instance that is stopped with \`appcfg stop\` (or via the Admin Console UI) has 30 seconds to finish handling requests before it is forcibly terminated.	Instances are created on demand to handle requests and automatically turned down when idle, based on the \`idle_timeout\` configuration

			parameter. As with manual scaling, an instance that is stopped with `appcfg stop` (or via the Admin Console UI) has 30 seconds to finish handling requests before it is forcibly terminated.
Instance Addressability	Instances are anonymous.	Instances are addressable at URLs with the form: `http://instance.version.module.app_id.appspot.com`. If you have set up a wildcard subdomain mapping for a custom domain, you can also address a module or any of its instances via a URL of the form `http://module.domain.com` or `http://instance.module.domain.com`. You can reliably cache state in each instance and retrieve it in subsequent requests.	Same as manual scaling.
Scaling	App Engine scales the number of instances automatically in response to processing volume. This scaling factors in the `automatic_scaling` settings that are provided on a per-version basis in the configuration file uploaded with the module version.	You configure the number of instances of each module version in that module’s configuration file. The number of instances usually corresponds to the size of a dataset being held in memory or the desired throughput for offline work.	A basic scaling module version is configured with a maximum number of instances using the `basic_scaling` setting's `max_instances` parameter. The number of live instances scales with the processing volume.
Free Daily Usage Quota	28 instance-hours	8 instance-hours	8 instance-hours

Instance classes

Instances are priced based on an hourly rate determined by the instance class.

Instance Class	Memory Limit	CPU Limit	Cost per Hour per Instance
B1	128 MB	600 Mhz	$0.05
B2	256 MB	1.2 Ghz	$0.10
B4	512 MB	2.4 Ghz	$0.20
B4_1G	1024 MB	2.4 Ghz	$0.30
B8	1024 MB	4.8 Ghz	$0.40
F1	128 MB	600 Mhz	$0.05
F2	256 MB	1.2 Ghz	$0.10
F4	512 MB	2.4 Ghz	$0.20
F4_1G	1024 MB	2.4 Ghz	$0.30

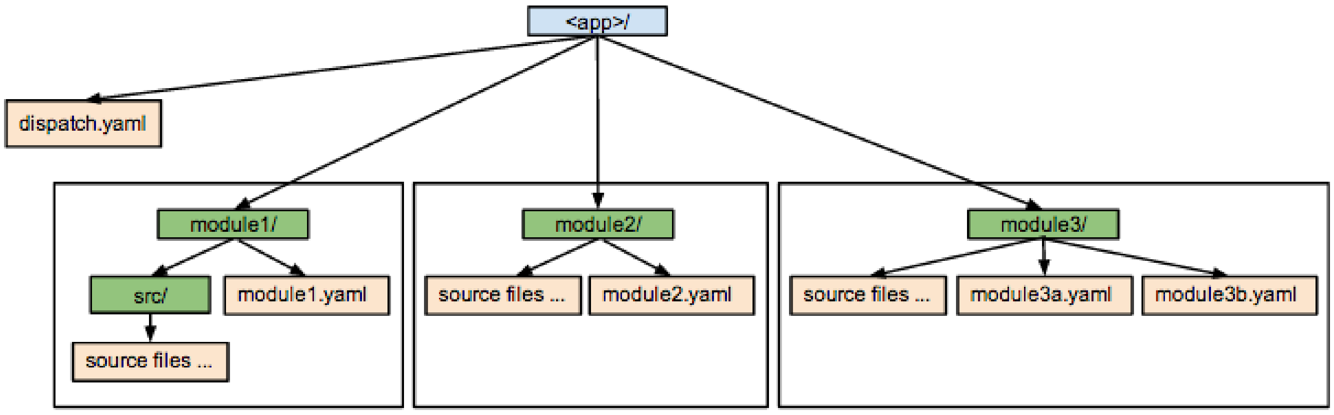
Manual and basic scaling instances are billed at hourly rates based on uptime. Billing begins when an instance starts and ends fifteen minutes after a manual instance shuts down or fifteen minutes after a basic instance has finished processing its last request. Runtime overhead is counted against the instance memory limit. This will be higher for Java than for other languages.

Important: When you are billed for instance hours, you will not see any instance classes in your billing line items. Instead, you will see the appropriate multiple of instance hours. For example, if you use an F4 instance for one hour, you do not see "F4" listed, but you will see billing for four instance hours at the F1 rate.

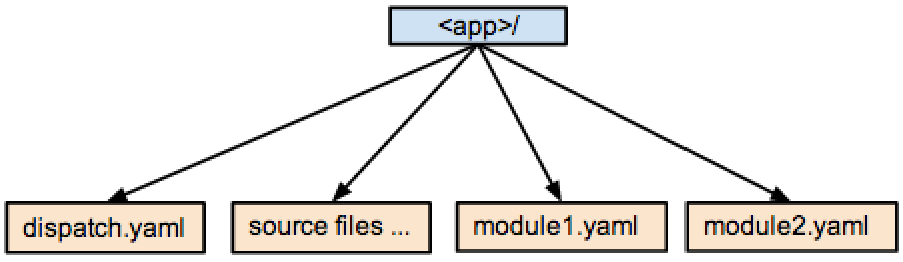
Configuration

Each version of a module is defined in a `.yaml` file, which gives the name of the module and version. The yaml file usually takes the same name as the module it defines, but this is not required. If you are deploying several versions of a module, you can create multiple yaml files in the same directory, one for each version.

Typically, you create a directory for each module, which contains the module's yaml file(s) 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 modules. In module1 the source files are contained in a subdirectory, in module2 they are at the same level as the yaml file; module3 has 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 module, you can explicitly include the parameter "module: default" or leave the module parameter out of the file.

Each module's configuration file defines the scaling type and instance class for a specific module/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.

For each module 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 Configsection. The following examples show how to configure modules for each scaling type.

Manual Scaling

```
application: simple-sample
module: my_module
version: uno
runtime: php
instance_class: B8
manual_scaling:
  instances: 5
```

`instance_class:`

The instance class size for this module. When using manual scaling, the B1, B2, B4, B4_1G, and B8 instance classes are available. If you do not specify a class, B2 is assigned by default.

`manual_scaling:`

Required to enable manual scaling for a module.

`instances:`

The number of instances to assign to the module at the start.

Basic Scaling

```
application: simple-sample
module: my_module
version: uno
runtime: php
instance_class: B8
basic_scaling:
  max_instances: 11
  idle_timeout: 10m
```

`instance_class:`

The instance class size for this module. When using basic scaling, the B1, B2, B4, B4_1G, and B8 instance classes are available. If you do not specify a class, B2 is assigned by default.

`basic_scaling:`

Required to enable basic scaling for a module.

`max_instances:`

Required. The maximum number of instances for App Engine to create for this module version. This is useful to limit the costs of a module.

`idle_timeout:`

Optional. The instance will be shut down this amount of time after receiving its last request.

Automatic Scaling

```
application: simple-sample
module: my_module
version: uno
runtime: php
instance_class: F2
automatic_scaling:
  min_idle_instances: 5
  max_idle_instances: automatic  # default value
  min_pending_latency: automatic  # default value
  max_pending_latency: 30ms
```

`instance_class:`

The Instance Class size for this module. When using automatic scaling, only the F1, F2, F4, and F4_1G instance classes are available. If you do not specify a class, F1 is assigned by default.

`automatic_scaling:`

Optional. Automatic scaling is assumed by default.

`min_idle_instances:`

The minimum number of idle instances that App Engine should maintain for this version. Only applies to the default version of a module, since other versions are not expected to receive significant traffic. Please keep in mind:

- A low minimum helps keep your running costs down during idle periods, but means that fewer instances may be immediately available to respond to a sudden load spike.

- A high minimum allows you to prime the application for rapid spikes in request load. App Engine keeps that number of instances in reserve at all times, so an instance is always available to serve an incoming request, but you pay for those instances. This functionality replaces the deprecated "Always On" feature, which ensured that a fixed number of instances were always available for your application. Once you've set the minimum number of idle instances, you can see these instances marked as "Resident" in the Instancestab of the Admin Console.

  If you set a minimum number of idle instances, pending latency will have less effect on your application's performance. Because App Engine keeps idle instances in reserve, it is unlikely that requests will enter the pending queue except in exceptionally high load spikes. You will need to test your application and expected traffic volume to determine the ideal number of instances to keep in reserve.

`max_idle_instances:`

The maximum number of idle instances that App Engine should maintain for this version. Please keep in mind:

- A high maximum reduces the number of idle instances more gradually when load levels return to normal after a spike. This helps your application maintain steady performance through fluctuations in request load, but also raises the number of idle instances (and consequent running costs) during such periods of heavy load.

- A low maximum keeps running costs lower, but can degrade performance in the face of volatile load levels.

Note: When settling back to normal levels after a load spike, the number of idle instances may temporarily exceed your specified maximum. However, you will not be charged for more instances than the maximum number you've specified.

`min_pending_latency`:

The minimum amount of time that App Engine should allow a request to wait in the pending queue before starting a new instance to handle it.

- A low minimum means requests must spend less time in the pending queue when all existing instances are active. This improves performance but increases the cost of running your application.
- A high minimum means requests will remain pending longer if all existing instances are active. This lowers running costs but increases the time users must wait for their requests to be served.

`max_pending_latency`:

The maximum amount of time that App Engine should allow a request to wait in the pending queue before starting a new instance to handle it.

- A low maximum means App Engine will start new instances sooner for pending requests, improving performance but raising running costs.
- A high maximum means users may wait longer for their requests to be served (if there are pending requests and no idle instances to serve them), but your application will cost less to run.

The default module

Every application must have a single default module. To define the default module, include the setting `module: default` in the module's yaml file, or leave the setting out.

Optional configuration files

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

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

If you would like to update these files automatically during each deployment, put them in the top level app directory and specify the app directory when you issue the `appcfg.py update` command.

If you place configuration files inside a module's directory (alongside the `app.yaml` file for the module), they will be updated only when you explicitly name that module's yaml file in the `update` command. They will not be updated when you update the root app directory that contains subfolders for each module.

You can also update configuration files individually using the special update commands (`update_dispatch`,`update_queues`, `update_indexes`, `update_cron`, `update_dos`) and specifying the app directory, or by just naming the files themselves in the `update` command.

An example

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

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

```
application: simple-sample

version: uno

runtime: php

api_version: 1

threadsafe: true
```

This configuration would create a default module with automatic scaling and a public address of`http://simple-sample.appspot.com`.

Next, assume that you want to create a module 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:

```
application: simple-sample

module: mobile-frontend

version: uno

runtime: php

api_version: 1

threadsafe: true

automatic_scaling:

  min_idle_instances: 2

  max_pending_latency: 1s
```

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

Finally, add a module, called `my-module` 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 module at any given time. Also, these jobs will need to handle a large amount of in-memory processing, thus you'll want modules with an increased memory configuration. To configure this you would create a `my-module.yaml` configuration file with the following contents.

```
application: simple-sample
module: my-module
version: uno
runtime: php
api_version: 1
threadsafe: true
instance_class: B8
manual_scaling:
  instances: 1
```

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

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

Uploading modules

To deploy the example above, use the appcfg update command. If you are uploading the app for the first time, the default module must be uploaded first, or if you are listing multiple modules, the default module must be the first module in the file list:

```
cd simple-sample

appcfg update app.yaml mobile-frontend.yaml my-module.yaml
```

You will receive verification via the command line as each module is successfully deployed.

Once the application has been successfully deployed you can access it at http://simple-sample.appspot.com. You can also access each of the modules individually:

- http://default.simple-sample.appspot.com
- http://mobile-frontend.simple-sample.appspot.com
- http://my-module.simple-sample.appspot.com

If you run multiple versions of a module, you can access a specific version by prepending the version name to the URI. For example, http://uno.default.simple-sample.appspot.com will target version uno of the default module. Routing Requests to Modules explains addressing module instances in detail.

Instance states

A manual or basic scaled instance can be in one of two states: Running or Stopped. All instances of a particular module/version share the same state. You can change the state of all the instances belonging to a module/version using the appcfg command or the Modules API.

Startup

Each module instance is created in response to a start request, which is an empty GET request to `/_ah/start`. App Engine sends this request to bring an instance into existence; users cannot send a request to `/_ah/start`. Manual and basic scaling instances must respond to the start request before they can handle another request. The start request can be used for two purposes:

•To start a program that runs indefinitely, without accepting further requests

•To initialize an instance before it receives additional traffic

Manual scaling instances and basic scaling instances startup differently. When you start a manual scaling instance, App Engine immediately sends a `/_ah/start` request to each instance. When you start an instance of a basic scaling module, App Engine allows it to accept traffic, but the `/_ah/start` request is not sent to an instance until it receives its first user request. Multiple basic scaling instances are only started as necessary, in order to handle increased traffic.

When an instance responds to the `/_ah/start` request with an HTTP status code of `200–299` or `404`, it is considered to have successfully started and can handle additional requests. Otherwise, App Engine terminates the instance. Manual scaling instances are restarted immediately, while basic scaling instances are restarted only when needed for serving traffic.

Instance uptime

App Engine attempts to keep manual and basic scaling instances running indefinitely. However, at this time there is no guaranteed uptime for manual and basic scaling instances. Hardware and software failures that cause early termination or frequent restarts can occur without prior warning and may take considerable time to resolve; thus, you should construct your application in a way that tolerates these failures. The App Engine team will provide more guidance on expected instance uptime as statistics become available.

Here are some good strategies for avoiding downtime due to instance restarts:

•Use load balancing across multiple instances.

•Configure more instances than are normally required to handle your traffic patterns.

•Write fall-back logic that uses cached results when a manual scaling instance is unavailable.

•Reduce the amount of time it takes for your instances to start up and shutdown.

•Duplicate the state information across more than one instance.

•For long-running computations, checkpoint the state from time to time so you can resume it if it doesn't complete.

It's also important to recognize that the shutdown hook is not always able to run before an instance terminates. In rare cases, an outage can occur that prevents App Engine from providing 30 seconds of shutdown time. Thus, we recommend periodically checkpointing the state of your instance and using it primarily as an in-memory cache rather than a reliable data store.

Monitoring resource usage

The Instances Console section of the Administration Console provides visibility into how instances are performing. By selecting your module and version in the dropdowns, you can see the memory and CPU usage of each instance, uptime, number of requests, and other statistics. You can also manually initiate the shutdown process for any instance.

You also can use the Runtime API to access statistics showing the CPU and memory usage of your instances. These statistics help you understand how resource usage responds to requests or work performed, and also how to regulate the amount of data stored in memory in order to stay below the memory limit of your instance class.

Logging

You can use the Logs API to access your app's request and application logs. In particular, the `fetch()` function allows you to retrieve logs using various filters, such as request ID, timestamp, module ID, and version ID.

Application (user/app-generated) logs are periodically flushed while manual and basic scaling instances handle requests; since modules can run on a request a long time, logs may not flush for a while. You can tune the flush settings, or force an immediate flush, using the Logs API. When a flush occurs, a new log entry is created at the time of the flush, containing any log messages that have not yet been flushed. These entries show up in the Logs Console marked with flush, and include the start time of the request that generated the flush.

Communication between modules

Modules can share state by using the Datastore and Memcache. They can collaborate by assigning work between them using Task Queues. To access these shared services, use the corresponding App Engine APIs. Calls to these APIs are automatically mapped to the application’s namespace.

The Modules API provides functions to retrieve the address of a module, a version, or an instance. This allows an application to send requests from one module, version, or instance to another module, version, or instance. This works in both the development and production environments. The Modules API also provides functions that return information about the current operating environment (module, version, and instance).

The following code sample shows how to get the module name and instance id for a request:

```
use google\appengine\api\modules\ModulesService;

$module = ModulesService::getCurrentModuleName();

$instance = ModulesService::getCurrentInstanceId();
```

The instance ID of an automatic scaled module will be returned as a unique base64 encoded value, e.g.`e4b565394caa`.

You can communicate between modules in the same app by fetching the hostname of the target module:

```
use google\appengine\api\modules\ModulesService;

$url = 'http://' . ModulesService::getHostname('my-backend') . '/';

$result = file_get_contents($url);
```

You can also use the URL Fetch service.

To be safe, the receiving module should validate that the request is coming from a valid client. You can check that the Inbound-AppId header or user-agent-string matches the app-id fetched with the AppIdentity service.

You can configure any manual or basic scaling module to accept requests from other modules in your app by adding the `"login:admin"` specification to the module's handler. In that case any URLFetch from any other module in the app will be automatically authenticated by App Engine, and any URLFetch call from a module that is not part of the application will be rejected.

If you want a module to receive requests from anywhere, you must code your own secure solution as you would for any App Engine application. This is usually done by implementing a custom API and authentication mechanism.

Limits

The maximum number of modules and versions that you can deploy depends on your app's pricing:

Limit	Free App	Paid App	Premier App
Maximum Modules Per App	5	20	20
Maximum Versions Per App	15	60	60

There is also a limit to the number of instances for each module with basic or manual scaling:

Maximum Instances per Manual/Basic Scaling Version		
Free App	US App (Paid/Premier)	EU App (Paid/Premier)
20	200	25

Routing Requests to Modules

HTTP requests from users can reach the appropriate module/version/instance in two ways: A request with a URL that ends at the domain level can be routed according to App Engine's default address routing rules. Alternatively, you can include a dispatch file that routes specific URL patterns according to your own rules.

If you test your app using the development server the available routing and dispatch features are slightly different. To programmatically create URLs that work with both production and development servers, use the Seerouting in the development server to learn more.

Addressing instances

You can target an HTTP request with varying degrees of specificity. In the following examples appspot.com can be replaced with your app's custom domain if you have one. The URL substrings "instance", "version", "module", and "app-id" represent application and module attributes that you have defined yourself.

These two address forms are guaranteed to reach their target (if it exists). They will never be intercepted and rerouted by a pattern in the dispatch file:

`http://instance.version.module.app-id.appspot.com`

> Sends a request to the named module, version, and instance. This address form can only be used by application administrators.

`http://version.module.app-id.appspot.com`

> Send the request to an available instance of the named module and version (round robin scheduling is used).

These address forms have a default routing behavior. Note that the default routing is overridden if there is a matching pattern in the dispatch file:

`http://module.app-id.appspot.com`

> Send the request to an available instance of the default version of the named module (round robin scheduling is used).

`http://version.app-id.appspot.com`

> Send the request to an available instance of the given version of the default module.

`http://app-id.appspot.com`

> Send the request to an available instance of the default version of the default module.

The default module is defined by explicitly giving a module the name "default," or by not including the name parameter in the module's config file. Requests that specify no module or an invalid module are routed to the default module. You can use the Admin Console to designate a default version for a module, when appropriate.

All modules are public by default, if you want to restrict access to a module, add the “login: admin” parameter to its handlers.

Please note that in April of 2013, Google stopped issuing SSL certificates for double-wildcard domains hosted at `appspot.com` (i.e. `*.*.appspot.com`). If you rely on such URLs for HTTPS access to your application, please change any application logic to use "-dot-" instead of ".". For example, to access version "1" of application "myapp" use "https://1-dot-myapp.appspot.com" instead of "https://1.myapp.appspot.com." If you continue to use "https://1.myapp.appspot.com" the certificate will not match, which will result in an error for any User-Agent that expects the URL and certificate to match exactly.

Dispatch file

You can create a dispatch file to override the default routing for URLs that do not specify a version (described above). This lets you route incoming requests to a specific module based on the path or hostname in the URL. For example, say that you want to route mobile requests like `http://simple-sample.appspot.com/mobile/` to a mobile frontend, route worker requests like `http://simple-sample.appspot.com/work/` to a static backend, and serve all static content from the default module.

To do this you can create a custom routing with a `dispatch.yaml` file.

```
dispatch:

  # Default module serves the typical web resources and all static resources.

  - url: "*/favicon.ico"
```

```
    module: default

  # Default module serves simple hostname request.

  - url: "simple-sample.appspot.com/"

    module: default

  # Send all mobile traffic to the mobile frontend.

  - url: "*/mobile/*"

    module: mobile-frontend

  # Send all work to the one static backend.

  - url: "*/work/*"

    module: static-backend
```

The dispatch file can contain up to 10 routing rules. When specifying the url string, neither the hostname nor the path can be longer than 100 characters.

As you can see, `dispatch.yaml` includes support for glob characters (however, yaml syntax requires that you include such expressions in quotes to denote they are strings). Glob characters can be used only before the hostname and at the end of the path. If you prefer general routing rules that match many possible requests, you could specify the following:

```
# Send any path that begins with “simple-sample.appspot.com/mobile” to the mobile-frontend module.

- url: "simple-sample.appspot.com/mobile*"

  module: mobile-frontend

# Send any domain/sub-domain with a path that starts with “work” to the static backend module.

- url: "*/work*"

  module: static-backend
```

You can also write expressions that are more strict:

```
# Matches the path "/fun", but not "/fun2" or "/fun/other"

- url: "*/fun"

  module: mobile-frontend

# Matches the hostname 'customer1.myapp.com', but not '1.customer1.myapp.com.

- url: "customer1.myapp.com/*"

  module: static-backend
```

The dispatch file can be uploaded together with your module files using a single appcfg update command. The names of all the modules used in the dispatch file must appear before the dispatch file itself. You can also upload the dispatch file separately, similarly making sure that all the modules mentioned in the file have already been uploaded.

Routing in the development server

Addressing instances

The development server creates all manual scaling instances at startup. Instances for automatic and basic scaling modules are managed dynamically. The server assigns a port to each module, so clients can depend on the server to load-balance and select an instance automatically. The port assignments for addressing each module appear in the server's log message stream. Here are the ports for an app that defines three modules, the scaling type of each module is not relevant:

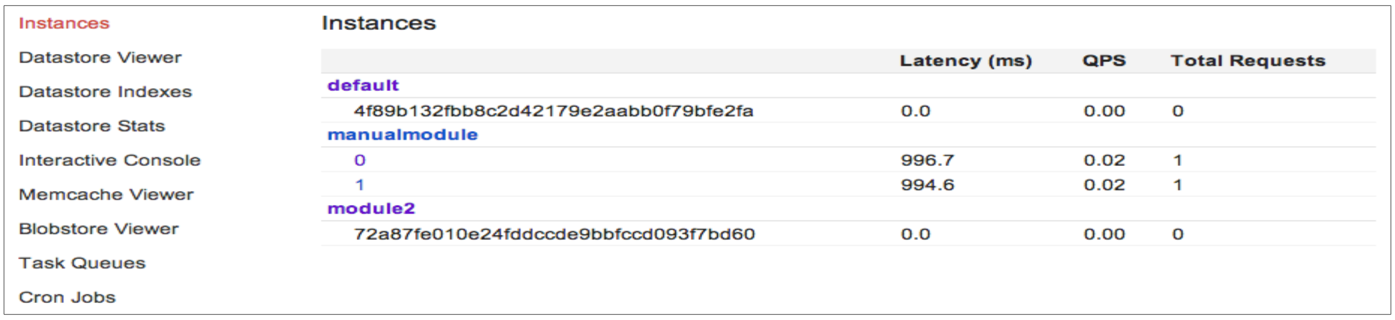
```
INFO Starting module "default" running at: http://localhost:8084

INFO Starting module "module1" running at: http://localhost:8082

INFO Starting module "module2" running at: http://localhost:8083
```

When you use a module's address (for example `http://localhost:8082/`), the server will select (or create) an instance of the module and send the request to that instance.

The server assigns unique ports to each instance of a module. To discover these ports you need to use the admin server. There is a unique port for the admin server, it appears in the message log:

```
INFO Starting admin server at: http://localhost:8000
```

This address takes you to the admin server console. From there you can click on Instances to see the dynamic state of your app's instances:



Instances

	Latency (ms)	QPS	Total Requests
default			
4f89b132fbb8c2d42179e2aabb0f79bfe2fa	0.0	0.00	0
manualmodule			
0	996.7	0.02	1
1	994.6	0.02	1
module2			
72a87fe010e24fddccde9bbfccd093f7bd60	0.0	0.00	0

A separate entry will appear for each manual and basic instance. The instance numbers are links with unique port addresses for each instance. You can hover over a link to see the port assigned to that instance, or click on the link to send a request directly to that instance.

Dispatch files

If your app uses a `dispatch.yaml` file, the log messages stream will

include a dispatcher port:

```
INFO Starting dispatcher running at: http://localhost:8080
```

Requests to this port will be routed according to the rules in the dispatch file. The server does not support`dispatch.yaml` file rules that include hostnames (for example, `url: "customer1.myapp.com/*"`). Rules with relative path patterns (`url: "*/fun"`) will work, so you can use URLs like`http://localhost/fun/mobile` to reach instances. The server will report an error in the log stream if you try to start an application with a `dispatch.yaml` file that contains host-based rules.

Dispatch files and cron files

A routing conflict can occur if your app has a dispatch file and a `cron.yaml` file, and the same URL path appears in both files. The cron file may specify a target module that is not the same as the module associated with the URL in the dispatch file. When the cron job runs, the URL is routed via the dispatch file, and the module specified there overrides the target in the cron file.