gevent-tools Documentation

Release 0.6.0

Jeff Lindsay

May 17, 2014

Release: v0.6.0 (*Installation*)
License: MIT

Ginkgo is a lightweight framework for writing network service daemons in Python. It currently focuses on gevent as its core networking and concurrency layer.

The core idea behind Ginkgo is the "service model", where your primary building block or component of applications are composable services. A service is a mostly self-contained module of your application that can start/stop/reload, contain other services, manage async operations, and expose configuration.

```
class ExampleService(Service):
    setting = Setting("example.setting", default="Foobar")

    def __init__(self):
        logging.info("Service is initializing.")

        self.child_service = AnotherService()
        self.add_service(self.child_service)

    def do_start(self):
        logging.info("Service is starting.")

        self.spawn(self.something_async)

    def do_stop(self):
        logging.info("Service is stopping.")

    def do_reload(self):
        logging.info("Service is reloading.")

    # ...
```

Around this little bit of structure and convention, Ginkgo provides just a few baseline features to make building both complex and simple network daemons much easier:

- Service class primitive for composing daemon apps from simple components
- Dynamic configuration loaded from regular Python source files
- Runner and service manager tool for easy, consistent usage and deployment
- Integrated support for standard Python logging

CHAPTER 1

User Guide

1.1 Introduction

1.1.1 Origin

Ginkgo evolved from a project called "gevent_tools" that started as a collection of common features needed when building gevent applications. The author had previously made a habit of building lots of interesting little servers as a hobby, and then at work found himself writing and dealing with lots more given the company's service oriented architecture. Accustomed to using the application framework in Twisted, when he finally saw the light and discovered gevent, there was no such framework for that paradigm.

Dealing with so many projects, it was not practical to reinvent the same basic features and architecture over and over again. The same way web frameworks made it easy to "throw together" a web application, there needed to be a way to quickly "throw together" network daemons. Not just simple one-off servers, but large-scale, complex applications – often part of a larger distributed system.

Through the experience of building large systems, a pattern emerged that was like a looser, more object-oriented version of the actor model based around the idea of services. This became the main feature of gevent_tools and it was later renamed gservice. However, with the hope of supporting other async mechanisms other than gevent's green threads (such as actual threads or processes, or other similar network libraries), the project was renamed Ginkgo.

1.1.2 Vision

The Ginkgo microframework is a minimalist foundation for building very large systems, beyond individual daemons. There were originally plans for gevent_tools to include higher-level modules to aid in developing distributed applications, such as service discovery and messaging primitives.

While Ginkgo will remain focused on "baseline" features common to pretty much all network daemons, a supplementary project to act as a "standard library" for Ginkgo applications is planned. Together with Ginkgo, the vision would be to quickly "throw together" distributed systems from simple primitives.

1.1.3 Inspiration

Most of Ginkgo was envisioned by taking good ideas from other projects, simplifying to their essential properties, and integrating them together. A lot of thanks goes out to these projects.

Twisted is the first great Python evented daemon framework. The two big ideas borrowed from Twisted are their application framework and twistd. They directly inspired the service model and the Ginkgo runner.

Trac is known for the problem it solves, and not so much for its great architecture. However, its component model and configuration API were a big influence on Ginkgo. Trac components are how we think of Ginkgo services, and the way Ginkgo defines configuration settings is directly inspired by the Trac configuration API.

These projects also had some influence on Ginkgo's design and philosophy: Gunicorn, Mongrel, Apache, Django, Flask, python-daemon, Diesel, Tornado, Erlang/OTP, Typeface, Akka, Configgy, Ostrich, and others.

1.2 Installation

Ginkgo is currently only available via GitHub, as it won't be released on PyPI until it reaches a stable 1.0 release.

1.2.1 Get the Code

You can either clone the public repository:

```
$ git clone git://github.com/progrium/ginkgo.git
```

Download the tarball:

```
$ curl -OL https://github.com/progrium/ginkgo/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/progrium/ginkgo/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

```
$ python setup.py install
```

1.3 Quickstart

Before you get started, be sure you have Ginkgo installed.

1.3.1 Hello World Service

The simplest service you could write looks something like this:

```
from ginkgo import Service

class HelloWorld(Service):
    def do_start(self):
        self.spawn(self.hello_forever)

    def hello_forever(self):
        while True:
            print "Hello World"
            self.async.sleep(1)
```

If you save this as *hello.py* you can run it with the Ginkgo runner:

```
$ ginkgo hello.HelloWorld
```

This should run your service, giving you a stream of "Hello World" lines.

To stop your service, hit Ctrl+C.

1.3.2 Writing a Server

A service is not a server until you make it one. Using gevent, this is easy using the StreamServer service to do the work of running a TCP server:

```
from ginkgo import Service
from ginkgo.async.gevent import StreamServer

class HelloWorldServer(Service):
    def __init__(self):
        self.add_service(StreamServer(('0.0.0.0', 7000), self.handle))

    def handle(self, socket, address):
        while True:
            socket.send("Hello World\n")
            self.async.sleep(1)
```

Save this as *quickstart.py* and run with:

```
$ ginkgo quickstart.HelloWorldServer
```

It will start listening on port 7000. We can connect with netcat:

```
$ nc localhost 7000
```

Again we see a stream of "Hello World" lines, but this time being sent over TCP. You can open more netcat connections to see it running concurrently just fine.

Notice our HelloWorldServer implementation is *composed* of a generic StreamServer and doesn't need to implement anything else other than a handler for that StreamServer.

1.3.3 Writing a Client

A client that maintains a persistent connection (or maybe pool of connections) to a server also makes sense to be modeled as a Service. Let's add a client to our HelloWorldServer in our quickstart module. Now it looks like this:

```
from ginkgo import Service
from ginkgo.async.gevent import StreamServer
from ginkgo.async.gevent import StreamClient

class HelloWorldServer(Service):
    def __init__(self):
        self.add_service(StreamServer(('0.0.0.0', 7000), self.handle))

    def handle(self, socket, address):
        while True:
            socket.send("Hello World\n")
            self.async.sleep(1)

class HelloWorldClient(Service):
    def __init__(self):
        self.add_service(StreamClient(('0.0.0.0', 7000), self.handle))
```

```
    def handle(self, socket):
        fileobj = socket.makefile()
        while True:
            print fileobj.readline().strip()
```

Save and run the server first with:

```
$ ginkgo quickstart.HelloWorldServer
```

Let that run, switch to a new terminal and run the client with:

```
$ ginkgo quickstart.HelloWorldClient
```

As you'd expect, the client connects to the server and prints all the "Hello World" lines it receives.

1.3.4 Service Composition

We've already been doing service composition by using generic TCP server and client services to build our HelloWorld services. These primitives are services themselves, just like the ones you've been making. So you can compose and aggregate your own services the same way.

Let's combine our client and server by add a HelloWorld service in our quickstart module. It now looks like this:

```
from ginkgo import Service
from ginkgo.async.gevent import StreamServer
from ginkgo.async.gevent import StreamClient

class HelloWorldServer(Service):
    def __init__(self):
        self.add_service(StreamServer(('0.0.0.0', 7000), self.handle))

    def handle(self, socket, address):
        while True:
            socket.send("Hello World\n")
            self.async.sleep(1)

class HelloWorldClient(Service):
    def __init__(self):
        self.add_service(StreamClient(('0.0.0.0', 7000), self.handle))

    def handle(self, socket):
        fileobj = socket.makefile()
        while True:
            print fileobj.readline().strip()

class HelloWorld(Service):
    def __init__(self):
        self.add_service(HelloWorldServer())
        self.add_service(HelloWorldClient())
```

Save and we can run our new aggregate service:

```
$ ginkgo quickstart.HelloWorld
```

Now the client and server are both running, giving us effectively what we came in with.

1.3.5 Using a Web Framework

Adding a web server our HelloWorld service is quite trivial. Here we use gevent's WSGI server implementation:

```
from ginkgo import Service
from ginkgo.async.gevent import StreamServer
from ginkgo.async.gevent import StreamClient
from ginkgo.async.gevent import WSGIServer

class HelloWorldServer(Service):
    def __init__(self):
        self.add_service(StreamServer(('0.0.0.0', 7000), self.handle))

    def handle(self, socket, address):
        while True:
            socket.send("Hello World\n")
            self.async.sleep(1)

class HelloWorldClient(Service):
    def __init__(self):
        self.add_service(StreamClient(('0.0.0.0', 7000), self.handle))

    def handle(self, socket):
        fileobj = socket.makefile()
        while True:
            print fileobj.readline().strip()

class HelloWorldWebServer(Service):
    def __init__(self):
        self.add_service(WSGIServer(('0.0.0.0', 8000), self.handle))

    def handle(self, environ, start_response):
        start_response('200 OK', [('Content-Type', 'text/html')])
        return ["<strong>Hello World</strong>"]

class HelloWorld(Service):
    def __init__(self):
        self.add_service(HelloWorldServer())
        self.add_service(HelloWorldClient())
        self.add_service(HelloWorldWebServer())
```

Running *quickstart.HelloWorld* with Ginkgo will run a server, a client, and a web server. The client will be printing our stream of "Hello World" lines. Our server is also available to be connected to via netcat. And we can also connect to our web server with curl:

```
$ curl http://localhost:8000
```

And we see a strong declaration of "Hello World".

In that example our web server implements a small WSGI application, but you can also use any WSGI compatible web framework. Here is an example of the Flask Hello World runnable with Ginkgo using *AppServer*:

```
from flask import Flask
from ginkgo.async.gevent import WSGIServer

app = Flask(__name__)

@app.route("/")
def hello():
```

```
    return "Hello World!"

def AppServer():
    return WSGIServer(('0.0.0.0', 8000), app)
```

Notice AppServer a callable that returns a service, in this case a pre-configured WSGIServer.

1.3.6 Using Configuration

TODO

1.4 Ginkgo Manual

1.4.1 Service Model

A service is an application component that starts and stops. It manages its own concurrency primitives and sub-service components. Creating a service just involves inheriting from the *Service* class and implementing any of the hooks needed of the service protocol:

`Service.`**`do_start`**`()`
: The do_start hook is where you implement what happens when the service is starting. Often this is where you bind to ports, open connectins, or spawn async loops. It should not be where actual service work is done, only the start up tasks.

  Before do_start is run, all child services are started, so you can assume they have been started. If you wish to make do_start run before child services, you can set the *start_before* class variable to True.

`Service.`**`do_stop`**`()`
: The do_stop hook is where you implement what happens when the service is stopping. This is often manipulating state in order to shutdown and clean up. It does not need to kill async operations or stop child services since that is taken care of by Ginkgo.

`Service.`**`do_reload`**`()`
: The do_reload hook is called when a parent service receives a reload call. In most cases, this is ultimately attached to SIGHUP signals. It is not meant to be a restart, which is a full stop and stop. Instead, you can use reload to reload state while running, such as configuration.

Otherwise, a class that inherits from *Service* is like any other class and should be thought of as the primary interface to the component it represents. If the code for a component is too much to live in one service class, it's good practice to split it into sub-component services. In these cases, the parent service often doesn't do any work itself, but is just a container and API facade for the child component services.

Here is a typical service implementation:

```
from ginkgo import Service

class MyService(Service):
    def __init__(self):
      self.subcomponent = SubcomponentService()
      self.add_service(self.subcomponent)

    def do_start(self):
```

TODO

1.4.2 Using Configuration

Add the `-h` argument flag to our runner call:

```
$ ginkgo service.HelloWorld -h
```

You'll see that the `ginkgo` runner command itself is very simple, but what's interesting is the last section:

```
config settings:
  daemon         True or False whether to daemonize [False]
  group          Change to a different group before running [None]
  logconfig      Configuration of standard Python logger. Can be dict for basicConfig,
                  dict with version key for dictConfig, or ini filepath for fileConfig. [None]
  logfile        Path to primary log file. Ignored if logconfig is set. [/tmp/HelloWorld.log]
  loglevel       Log level to use. Valid options: debug, info, warning, critical
                  Ignored if logconfig is set. [debug]
  pidfile        Path to pidfile to use when daemonizing [None]
  rundir         Change to a directory before running [None]
  umask          Change file mode creation mask before running [None]
  user           Change to a different user before running [None]
```

These are builtin settings and their default values. If you want to set any of these, you have to create a configuration file. But you can also create your own settings, so let's first change our Hello World service to be configurable:

```
from ginkgo import Service, Setting

class HelloWorld(Service):
    message = Setting("message", default="Hello World",
        help="Message to print out while running")

    def do_start(self):
        self.spawn(self.message_forever)

    def message_forever(self):
        while True:
            print self.message
            self.async.sleep(1)
```

Running `ginkgo service.HelloWorld -h` again should now include your new setting. Let's create a configuration file now called *service.conf.py*:

```
import os
daemon = bool(os.environ.get("DAEMONIZE", False))
message = "Services all the way down."
service = "service.HelloWorld"
```

A configuration file is simply a valid Python source file. In it, you define variables of any type using the setting name to set them.

There's a special setting calling `service` that must be set, which is the class path target telling it what service to run. To run with this configuration, you just point `ginkgo` to the configuration file:

```
$ ginkgo service.conf.py
```

And it should start and you should see "Services all the way down" repeating.

You don't have direct access to set config settings from the `ginkgo` tool, but you can set values in your config to pull from the environment. For example, our configuration above lets us force our service to daemonize by setting the `DAEMONIZE` environment variable:

```
$ DAEMONIZE=yes ginkgo service.conf.py
```

To stop the daemonized process, you can manually kill it or use the service management tool `ginkgoctl`:

```
$ ginkgoctl service.conf.py stop
```

1.4.3 Service Manager

Running and stopping your service is easy with `ginkgo`, but once you daemonize, it gets harder to interface with it. The `ginkgoctl` utility is for managing a daemonized service process.

```
$ ginkgoctl -h
usage: ginkgoctl [-h] [-v] [-p PID]
                 [target] {start,stop,restart,reload,status,log,logtail}

positional arguments:
  target                service class path to use (modulename.ServiceClass) or
                        configuration file path to use (/path/to/config.py)
  {start,stop,restart,reload,status,log,logtail}

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  -p PID, --pid PID     pid or pidfile to use instead of target
```

Like `ginkgo` it takes a target class path or configuration file. For `stop`, `reload`, and `status` it can also just take a pid or pidfile with the `pid` argument.

Using `ginkgoctl` will always force your service to daemonize when you use the `start` action.

1.4.4 Service Model and Reloading

Our service model lets you implement three main hooks on services: `do_start()`, `do_stop()`, and `do_reload()`. We've used `do_start()`, which is run when a service is starting up. Not surprisingly, `do_stop()` is run when a service is shutting down. When is `do_reload()` run? Well, whenever `reload()` is called. :)

Services are designed to contain other services like object composition. Though after adding services to a service, when you call any of the service interface methods, they will propogate down to child services. This is done in the actual `start()`, `stop()`, and `reload()` methods. The `do_` methods are for you to implement specifically what happens for *that* service to start/stop/reload.

So when is `reload()` called? Okay, I'll skip ahead and just say it gets called when the process receives a SIGHUP signal. As you may have guessed, for convenience, this is exposed in `ginkgoctl` with the `reload` action.

The semantics of `reload` are up to you and your application or service. Though one thing happens automatically when a process gets a reload signal: configuration is reloaded.

One use of `do_reload()` is to take new configuration and perform any operations to apply that configuration to your running service. However, as long as you access a configuration setting by reference via the `Setting` descriptor, you may not need to do anything – the value will just update in real-time.

Let's see this in action. We'll change our Hello World service to have a `rate_per_minute` setting that will be used for our delay between messages:

```
from ginkgo import Service, Setting

class HelloWorld(Service):
    message = Setting("message", default="Hello World",
        help="Message to print out while running")

    rate = Setting("rate_per_minute", default=60,
        help="Rate at which to emit message")

    def do_start(self):
        self.spawn(self.message_forever)

    def message_forever(self):
        while True:
            print self.message
            self.async.sleep(60.0 / self.rate)
```

The default is 60 messages a minute, which results in the same behavior as before. So let's change our configuration to use a different rate:

```
import os
daemon = bool(os.environ.get("DAEMONIZE", False))
message = "Services all the way down."
rate_per_minute = 180
service = "service.HelloWorld"
```

Use `ginkgo` to start the service:

```
$ ginkgo service.conf.py
```

As you can see, it's emitting messages a bit faster now. About 3 per second. Now while that's running, open the configuration file and change rate_per_minute to some other value. Then, in another terminal, change to that directory and reload:

```
$ ginkgoctl service.conf.py reload
```

Look back at your running service to see that it's now using the new emit rate.

1.4.5 Using Logging

Logging with Ginkgo is based on standard Python logging. We make sure it works with daemonization and provide Ginkgo-friendly ways to configure it with good defaults. We even support reloading logging configuration.

Out of the box, you can just start logging. We encourage you to use the common convention of module level loggers, but obviously there is a lot of freedom in how you use Python logging. Let's add some logging to our Hello World, including changing our print call to a logger call as it's better practice:

```
import logging
from ginkgo import Service, Setting

logger = logging.getLogger(__name__)

class HelloWorld(Service):
    message = Setting("message", default="Hello World",
        help="Message to print out while running")

    rate = Setting("rate_per_minute", default=60,
        help="Rate at which to emit message")
```

```
    def do_start(self):
        logger.info("Starting up!")
        self.spawn(self.message_forever)

    def do_stop(self):
        logger.info("Goodbye.")

    def message_forever(self):
        while True:
            logger.info(self.message)
            self.async.sleep(60.0 / self.rate)
```

Let's run it with our existing configuration for a bit and then stop:

```
$ ginkgo service.conf.py
Starting process with service.conf.py...
2012-04-28 17:21:32,608    INFO service: Starting up!
2012-04-28 17:21:32,608    INFO service: Services all the way down.
2012-04-28 17:21:33,609    INFO service: Services all the way down.
2012-04-28 17:21:34,610    INFO service: Services all the way down.
2012-04-28 17:21:35,714    INFO service: Goodbye.
2012-04-28 17:21:35,714    INFO runner: Stopping.
```

Running `-h` will show you that the default logfile is going to be */tmp/HelloWorld.log*, which logging will create and append to if you daemonize.

To configure logging, Ginkgo exposes two settings for simple case configuration: `logfile` and `loglevel`. If that's not enough, you can use `logconfig`, which will override any value for `logfile` and `loglevel`.

Using `logconfig` you can configure logging as expressed by `logging.basicConfig`. By default, if you set `logconfig` to a dictionary, it will apply those keyword arguments to `logging.basicConfig`. You can learn more about `logging.basicConfig` here.

For advanced configuration, we also let you use `logging.config` from the `logconfig` setting. If `logconfig` is a dictionary with a `version` key, we will load it into `logging.config.dictConfig`. If `logconfig` is a path to a file, we load it into `logging.config.fileConfig`. Both of these are ways to define a configuration structure that lets you create just about any logging configuration. Read more about `logging.config` here.

1.5 Advanced Usage and Patterns

1.5.1 Service State Machine

TODO

1.5.2 Service Factory in Config

TODO

1.5.3 Using Configuration Groups

TODO

1.5.4 Using ZeroMQ

TODO

1.5.5 Async Backends

TODO

CHAPTER 2

API Reference

CHAPTER 3

Developer Guide