Faster Web Applications with SCGI

Jeroen Vermeulen

Abstract

Speed up your Web applications with SCGI.

If you're operating a Web server, chances are, you're not merely serving up static text and images. You're likely to be running some Web applications as well, where pages are generated on the fly by some program or script using CGI (Common Gateway Interface). Think of blogging software, bug trackers, news sites and content management systems—anything that turns the browser from a document viewer into a user interface. And, you probably write or at least tweak some of these yourself.

This article shows how to build faster Web applications using an alternative to CGI called SCGI (Simple Common Gateway Interface). SCGI is a protocol, not just a program, but its authors also provide a reference implementation, which is what we use here. It includes modules to use SCGI from Apache or lighttpd and Python classes to help you create SCGI applications. Implementations in other languages are available, but we examine the combination of Apache 2.x and Python here.

Where Does the Time Go?

Normally, a Web application runs briefly, but very frequently, in child processes of the Web server. When a client requests a page, the Web server consults its configuration and finds that the request should go to the application. It delegates the request to a child process, which in turn loads and runs the application program. The program may be a binary or a script in Perl, Python or PHP, shell commands, or just about anything else. The CGI standard defines how the program receives details about the request, including requested URL, requested body, authenticated user identity and originating IP address. The program reads these, produces a page in answer to the client's request, and exits. All this happens again at the next request.

Loading, running and exiting programs can be costly. It does make sense for sloppy programs: they may use memory without ever freeing it up again, for instance. In that case, you want the program to run briefly and then let the operating system clean up after it. But, with today's popular languages—Perl, Python, PHP, Java and shell scripts—there really aren't many problems with this. A well-written application really should be able to handle multiple requests in a single run.

Faster Service with SCGI

SCGI lets your program start once and continue servicing requests for as long as it likes. It works like this: a separate server process, called an SCGI server, runs separately from the Web server and manages one Web application. The Web server forwards all requests for that application to the application's SCGI server. It passes on details about the request in much the same form as in regular CGI.

The SCGI server delegates the request to a child process, just like the Web server did with a regular CGI application. The child process also runs the application, but that's where the similarity ends. Instead of exiting after it's done with that one request, the application can sit and wait for a new one. Each of the SCGI server's child processes runs one instance of the application, each sleeping until there is work for it to do.

The SCGI server spawns a new child process when none are available to take on the latest request—up to a configurable maximum, of course. It also cleans up crashing or exiting child processes, so your Web application can still bail out if things go wrong. But, most of the time, when a request arrives, the application is ready and waiting for it. That's why Ruby on Rails, the Web application framework, comes with the option to run on SCGI; it would be too slow otherwise.

Other Advantages

If the speedup isn't enough for you, there's more. The SCGI server process can be running on the same system as the Web server, but it doesn't have to be. You can offload the server by delegating some Web applications to separate systems, preferably behind a firewall where only the Web server can access them.

Even with just a single server, you can use SCGI to contain vulnerabilities. A normal CGI application starts out running under the same user identity as the Web server process. If an attacker manages to subvert a normal CGI application, your entire Web site may be at risk. An SCGI server, on the other hand, can run under its own user identity, so it can't easily affect the Web server or other applications even if it does run amok. Conversely, you don't need to give the Web server access to the application's code or data.
anymore; only the application as run by the SCGI server needs access. Everyone else must go through the Web server, which in turn
talks to the SCGI server.

You also can run an application in a chroot environment or a virtualized server. With CGI, that quickly becomes expensive and hard
to manage. When using SCGI, you start only one server process in your isolated environment—whether it's a chroot jail, a virtualized
server, a different user identity or another machine—and the entire application will stay there.

**Installing SCGI**

You need two components: the Python classes for building SCGI applications and a module for your Web server to make it “speak
SCGI” to the applications. If you use Red Hat package management (RPM), you can install these using `yum install python-scgi
apache2-mod_scgi`; users of Debian's apt can use `apt-get install python-scgi libapache2-mod-scgi`.

You also can install either component by hand. The Apache module requires a C compiler and Apache's `apxs` script. Some
distributions keep `apxs` in a separate development package rather than installing it as part of the regular Apache package.

Assuming you now have those components, next download the source tarball `scgi-1.12.tar.gz`, and run the commands shown in Listing
1.

**Listing 1. Installing SCGI by Hand**

```bash
# Unpack source directory scgi-1.12 from tarball
tar xzf scgi-1.12.tar.gz
cd scgi-1.12
# Build the Python part
python setup.py build
# Install Python module; we'll need root privileges
sudo python setup.py install
# Now build and install the Apache module
cd apache2
sudo make install
# Enable the SCGI module in Apache. This may fail,
# depending on your Apache version, but no matter.
sudo a2enmod scgi
# Make Apache's new configuration take effect
sudo /etc/init.d/apache2 force-reload
```

**Test Run**

Now, let’s make sure it all works. The Python package is a module with some classes, and normally, you'd write your application as a
program that imports that module. For debugging, however, you also can run it as a standalone application. When it receives a request
from the Web server, it simply prints the request's details as a text page. Perfect for a first test—no coding required!

Find the `scgi_server.py` module on your system. It should be installed in `/usr/lib/python2.4/site-packages/scgi` (the 2.4 may be 2.3 or
2.5 on your system). Then, run the module:

```bash
cd /usr/lib/python2.4/site-packages/scgi
python scgi_server.py
```

This listens for requests from the Web server on a TCP port on your system, using port 4000 by default. You can make it listen on a
different port by passing the desired port number as a command-line argument, such as: Garrick, shrink below (one line).

```bash
python /usr/lib/python2.4/site-packages/scgi/scgi_server.py 63000
```

The module keeps running until you kill it, so start it in a separate shell. Remember, you don't need to run an SCGI server as root or
even under the Web server's identity.

Now that the SCGI application is waiting for requests, pick a location on your Web site to delegate to the application. Let's say you
want it to answer all requests for “/scgitest” on this server. Write an Apache configuration snippet, as shown in Listing 2, to a new file
in `/etc/apache2/conf.d`. Garrick, small font in listing 2.

**Listing 2. Apache Configuration Snippet**

```bash
# Load the SCGI module. This is really only needed
# if you installed manually and the "a2enmod scgi"
# command failed.
LoadModule scgi_module /usr/lib/apache2/modules/mod_scgi.so
```
The SCGI server doesn't really need to run on the same machine as the Web server, as you can see here. Simply make sure that the SCGI server's port is properly firewalled, so that only your Web server can reach it! That way, your application can be sure that all CGI parameters have been validated by the Web server first. If an attacker could connect directly to your SCGI application, you wouldn't be able to trust that information. The CGI parameter AUTHENTICATED_USER, for instance, tells your application that the request comes from a particular logged-in user. You can believe that only if you hear it from a properly configured Web server.

Make Apache reload its configuration with `sudo /etc/init.d/apache2 reload`. Your server should now serve a new location, /scgitest, that simply prints your request's CGI parameters when you access it. Verify this by looking it up in a browser. If your server's address is example.org, point your browser at http://example.org/scgitest. You should see a page that looks like Listing 3.

Listing 3. scgi_server.py returns request details.

```
9310l3.qrk
SERVER_SOFTWARE: 'Apache'
SCRIPT_NAME: '/scgitest'
REQUEST_METHOD: 'GET'
SERVER_PROTOCOL: 'HTTP/1.1'
QUERY_STRING: ''
CONTENT_LENGTH: '0'
HTTP_ACCEPT_CHARSET: 'UTF-8,*'
HTTP_USER_AGENT: 'Mozilla/5.0'
SERVER_NAME: 'testserver.example.org'
REMOTE_ADDR: '10.99.11.99'
SERVER_PORT: '80'
SERVER_ADDR: '192.0.34.166'
DOCUMENT_ROOT: '/srv/www/
SERVER_ADMIN: 'webmaster@example.org'
HTTP_HOST: 'testserver.example.org'
REQUEST_URI: '/scgitest'
HTTP_ACCEPT: 'text/html,text/plain,*/*;q=0.5'
REMOTE_PORT: '47088'
HTTP_ACCEPT_LANGUAGE: 'en'
SCGI: '1'
HTTP_ACCEPT_ENCODING: 'gzip,deflate'
```

If that's not what you see, take a look at the shell where you ran the module. It may have printed some helpful error message there. Or, if there is no reaction from the SCGI server whatsoever, the request may not have reached it in the first place; check the Apache error log.

Once you have this running, congratulations—the worst is behind you. Stop your SCGI server process so it doesn't interfere with what we're going to do next.

Writing an Application

Now, let's write a simple SCGI application in Python—one that prints the time.

We import the SCGI Python modules, then write our application as a handler for SCGI requests coming in through the Web server. The handler takes the form of a class that we derive from SCGIHandler. Call me unimaginative, but I've called the example handler class TimeHandler. We'll fill in the actual code in a moment, but begin with this skeleton:

```
#!/usr/bin/python
import scgi
import scgi.scgi_server

class TimeHandler(scgi.scgi_server.SCGIHandler):
    pass  # (no code here yet)

# Main program: create an SCGIServer object to
# listen on port 4000. We tell the SCGIServer the
# handler class that implements our application.
server = scgi.scgi_server.SCGIServer(}
```
handler_class=TimeHandler,
port=4000
)
# Tell our SCGIServer to start servicing requests.
# This loops forever.
server.serve()

You may think it strange that we must pass the SCGIServer our handler class, rather than a handler object. The reason is that server object will create handler objects of our given class as needed.

This first incarnation of TimeHandler is still essentially the same as the original SCGIHandler, so all it does is print out request parameters. To see this in action, try running this program and opening the scgitest page in your browser as before. You should see something like Listing 3 again.

Now, we want to print the time in a form that a browser will understand. We can't simply start sending text or HTML; we first must emit an HTTP header that tells the browser what kind of output to expect. In this case, let's stick with simple text. Add the following near the top of your program, right above the TimeHandler class definition:

```python
import time
def print_time(outfile):
    # HTTP header describing the page we're about
    # to produce. Must end with double MS-DOS-style
    # "CR/LF" end-of-line sequence. In Python, that
    # translates to "\r\n".
    outfile.write("Content-Type: text/plain\r\n\r"
    # Now write our page: the time, in plain text
    outfile.write(time.ctime() + "\n")
```

By now, you're probably wondering how we will make our handler class call this function. With SCGI 1.12 or newer, it's easy. We can write a method TimeHandler.produce() to override SCGIHandler's default action:

```python
class TimeHandler(scgi.scgi_server.SCGIHandler):
    # (remove the "pass" statement--we've got real
    # code here now)
    # This is where we receive requests:
    def produce(self, env, bodysize, input, output):
        # Do our work: write page with the time to output
        print_time(output)
```

We ignore them here, but produce() takes several arguments: env is a dict mapping CGI parameter names to their values. Next, bodysize is the size in bytes of the request body or payload. If you're interested in the request body, read up to bodysize bytes from the following argument, input. Finally, output is the file that we write our output page to.

If you have SCGI 1.11 or older, you need some wrapper code to make this work. In these older versions, you override a different method, SCGIHandler.handle_connection(), and do more of the work yourself. Simply copy the boilerplate code from Listing 4 into the TimeHandler class. It will set things up right and call produce(), so nothing else changes, and we can write produce() exactly as if we had a newer version of SCGI.

### Listing 4. Boilerplate Code for SCGI 1.11 or Older

```python
# Insert this definition into your handler class:
class TimeHandler(scgi.scgi_server.SCGIHandler):
    # ...
    def handle_connection(self, conn):
        input = conn.makefile("r")
        output = conn.makefile("w")
        env = self.read_env(input)
        bodysize = int(env.get("CONTENT_LENGTH",0))
        try:
            self.produce(env, bodysize, input, output)
        finally:
            output.close()
            input.close()
            conn.close()
```

Once again, run the application and check that it shows the time in your browser.

Next, to make things more interesting, let's pass some arguments to the request and have the program process them. The convention for arguments to Web applications is to tack a question mark onto the URL, followed by a series of arguments separated by
ampersands. Each argument is of the form name=value. If we wanted to pass the program a parameter called pizza with the value hawaii, and another one called drink with the value beer, our URL would look something like http://example.org/scgitest?pizza=hawaii&drink=beer.

Any arguments that the visitor passes to the program end up in the single CGI parameter QUERY_STRING. In this case, the parameter would read "pizza=hawaii&drink=beer". Here's something our TimeHandler might do with that: Garrick, shrink below.

class TimeHandler(scgi.scgi_server.SCGIHandler):
    def produce(self, env, bodysize, input, output):
        # Read arguments
        argstring = env['QUERY_STRING']
        # Break argument string into list of
        # pairs like "name=value"
        arglist = argstring.split('&')
        # Set up dictionary mapping argument names
        # to values
        args = {}
        for arg in arglist:
            (key, value) = arg.split('=')
            args[key] = value
        # Print time, as before, but with a bit of
        # extra advice
        print_time(output)
        output.write("Time for a pizza. I'll have the %s and a swig of %s!\n" %
                      (args['pizza'], args['drink']))

Now the application we wrote will not only print the time, but also suggest a pizza and drink as passed in the URL. Try it! You also can experiment with the other CGI parameters in Listing 3 to find more things your SCGI applications can do.

**Porting Applications**

Once you're comfortable writing programs using SCGI, you may want to try adapting existing applications to use it. Some well-known Web applications, such as MoinMoin (a wiki) and Trac (a wiki-based collaborative development environment), are implemented as Python modules. Both of these examples come with CGI scripts in Python that can be called from Apache. The CGI scripts are very short; they really don't do anything except import the application's modules and invoke a function on them.

If you find an application like that, all you really need to do to make it work with SCGI is take that little bit of Python code and move it into a produce() method, as in the examples you've seen here. If you have SCGI 1.12 or newer, you also might want to take a look at an alternative SCGIHandler method, produce_cgilike().

**Conclusion**

That's about all we have room for. If you wonder about how the CGI parameters work, try looking at the CGI standard, which calls them "request meta-variables" (see Resources).

Finally, a word of warning. You'll notice that the last example program dies horribly if you fail to pass the expected arguments. The SCGI server replaces the failing processes, so in this case, there's no real problem. But, this should remind you how careful you need to be when writing Web applications. Never trust the input you receive from outside! If a program can be crashed, someone can probably subvert it or take it out of action. People all over the world do that sort of thing for fun or profit, so take the risk seriously.

**Resources**

SCGI Downloads: [http://quixote.python.ca/releases](http://quixote.python.ca/releases)


Perl Interface: [http://search.cpan.org/~vipercode/SCGI/lib/SCGI.pm](http://search.cpan.org/~vipercode/SCGI/lib/SCGI.pm)

Lisp Interface: [http://randallsquared.com/download/scgi](http://randallsquared.com/download/scgi)
Trac: http://trac.edgewall.com
MoinMoin: http://moinmoin.wikiwikiweb.de