Member-only story

Write Your Own HTTP Server (Part 3)

Make your HTTP server WSGI compliant

Vaibhav Sinha · Follow
Published in Better Programming
7 min read · Jan 3, 2020

Listen Share More

Photo by Ilya Pavlov on Unsplash

This is the third piece in the series "Write Your Own HTTP Server." You can find Part 1 and Part 2 here.

In the last piece, we had built a working HTTP server that could accept connections from a client, parse the request, and send back a dummy response. The only thing that remains is to be able to load an application developed by the end user of our server so that we invoke the request handlers of the application and get a useful response back.

Loading someone's application should not be that hard. We have `importlib` available in Python, which lets us import the user's application dynamically at run time. The user just needs to tell us where the application code is available on disk. That can be done using the config file. The harder part is knowing how to communicate with that application, i.e. which function to invoke, with what parameters, and then what kind of reply to expect. This is where WSGI comes in.

Web Server Gateway Interface

When we develop web applications, we want to be able to deploy them on any server of our choice. And later we might decide to choose a totally different web server. Doing so should not require us to make any changes to our application. Our application should be oblivious to which server will be used to host it. This is only possible if all the servers decide to communicate with the applications in the same way. And that is what the Web Server Gateway Interface (WSGI) tries to achieve.

WSGI defines how any web server is going to invoke the application, what parameters it will send in, and what outcome it expects. PEP 3333, the specification for WSGI, describes it in detail, along with code samples for implementation. We'll discuss the specification in brief, and interested readers can read all the details from the PEP.

WSGI requires the imported application object to be a callable. Hence it can be a function, a method, or a class or object with `__call__` method defined. Every time a request comes in, the server is going to invoke this callable.

```
result = application(environ, start_response)
```

While invoking it, it's going to pass in two arguments:

- **The environment:** This would be a dictionary that contains CGI-style environment variables describing the HTTP request. Some of the keys of this dictionary would be `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, `QUERY_STRING`, `CONTENT_TYPE`, and `CONTENT_LENGTH`. This is how the parsed HTTP request is sent over to the application. There are also going to be WSGI-specific keys, such as `wsgi.version`, `wsgi.url_scheme`, `wsgi.input`, `wsgi.multithread`, etc. This extra metadata is for the application to know how it will be invoked. For example, if `wsgi.multithread` is set to *True*, then the application will be invoked by multiple threads at the same time, and hence would need to be thread-safe.
- **A callback function:** The second argument provided is a callable. The application must invoke it once it wants to start sending the response. The first argument the application will send to it will be the HTTP response status code, such as *200 OK*. The second argument is the dictionary of response headers to send back to the client. The third argument has to do with error handling, and we won't be discussing that here.

```
start_response(status, response_headers, exc_info=None)
```

As the result of the invocation, the application must return an iterable of bytestrings. This will be the response body sent back to the client. The server needs to iterate over the returned iterable and send the bytestrings on to the socket. When the iteration completes, the response is over, and the server may close the connection to the client.

Implementation

Now that we understand the interface we need to implement, let's get to it.

```
1  import logging
2  import queue
3  import threading
4  from typing import Type
5
6  from http_parser.pyparser import HttpParser
7
```

```
7
8   from gateway import WSGI
9
10  logger = logging.getLogger(__name__)
11
12
13  class Worker:
14
15      is_stopped = False
16
17      def __init__(self):
18          self.config = None
19          self.queue = None
20          self.gateway = None
21          self.kill_pill = None
22
23      def setup(self, config):
24          self.config = config
25
26          # What should be the size of this queue?
27          self.queue = queue.Queue(maxsize=config.get('concurrency', 10))
28          self.gateway = WSGI(config)
29          self.kill_pill = threading.Event()
30
31          threads = [RequestProcessorThread(name=f'RequestProcessor {i}', queue=self.queu
32          for t in threads:
33              t.start()
34
35      def run(self, listener):
36          logger.info("Accepting connections now")
37          while not self.is_stopped:
38              sock, _ = listener.accept()
39              self.submit(sock)
40
41      def submit(self, sock):
42          try:
43              self.queue.put(sock, timeout=self.config.get('timeout', 5))
44          except queue.Full:
45              # What should we do here?
46              pass
47
48      def shutdown(self):
49          self.is_stopped = True
50          self.kill_pill.set()
51
52
53  class RequestProcessorThread(threading.Thread):
54
```

```
55      def __init__(self, group=None, target=None, name=None, queue:queue.Queue=None, kill
56          super().__init__(group=group, target=target, name=name, args=args, kwargs=kwarg
57          self.queue: Type[queue.Queue] = queue
58          self.kill_pill = kill_pill
59          self.gateway = gateway
60
61      def run(self) -> None:
62          logger.info(f"Running thread {self.name}")
63          while not self.kill_pill.is_set():
64              try:
65                  socket = self.queue.get(block=True, timeout=1)
66                  self.process(socket)
67              except queue.Empty:
68                  continue
69
70      def process(self, sock):
71          p = HttpParser()
72          while True:
73              data = sock.recv(1024)
74              if not data:
75                  # The client closed the connection. Nothing to do anymore
76                  return
77
78              p.execute(data, len(data))
79              if p.is_message_complete():
80                  break
81
82          def write(data):
83              sock.send(data)
84
85          self.gateway.process(p, write)
86          sock.close()
```

Worker that delegates requests to WSGI class

I have changed the worker to have an instance of class WSGI that we're going to create next. This class is going to implement the WSGI interface and load the user's application based on the configuration provided. When a new request arrives, the server is going to parse it and then delegate it to this class. It is the responsibility of the WSGI class to then delegate it to the loaded application. Let's look at the WSGI class implementation.

```
1   import sys
2   from io import StringIO
3
```

```
4
5   class WSGI:
6
7       def __init__(self, config):
8           import importlib.util
9
10          self.config = config
11          app_loc, app_module, app = config['app-loc'], config['app-module'], config['app
12          spec = importlib.util.spec_from_file_location(app_module, app_loc + f"/{app_mod
13          module = importlib.util.module_from_spec(spec)
14          spec.loader.exec_module(module)
15
16          self.app = getattr(module, app)
17
18      def process(self, parsed_request, write):
19          body = []
20          if parsed_request.is_partial_body():
21              body.append(parsed_request.recv_body())
22
23          body = StringIO("".join(body))
24
25          env = {
26              'REQUEST_METHOD': parsed_request.get_method(),
27              'SCRIPT_NAME': '',
28              'PATH_INFO': parsed_request.get_path(),
29              'SERVER_NAME': self.config['address'],
30              'SERVER_PORT': self.config['port'],
31              'SERVER_PROTOCOL': 'HTTP/1.1',
32              'wsgi.version': (1, 0),
33              'wsgi.url_scheme': 'http',
34              'wsgi.input': body,
35              'wsgi.errors': sys.stderr,
36              'wsgi.multithread': True,
37              'wsgi.multiprocess': True,
38              'wsgi.run_once': False
39          }
40
41          for header, value in parsed_request.get_headers().items():
42              env[f'HTTP_{header}'] = value
43
44          headers = []
45
46          def start_response(status, response_headers, exc_info=None):
47              nonlocal headers
48              headers = [status, response_headers]
49
50          outputs = self.app(env, start_response)
51
```

```
52          status, response_headers = headers
53          write(str.encode('HTTP/1.1: %s\r\n' % status))
54          for header in response_headers:
55              write(str.encode('%s: %s\r\n' % header))
56          write(str.encode('\r\n'))
57
58          for output in outputs:
59              write(output)
```

WSGI implementation

As can be seen, WSGI class uses `importlib` to import the application using the details provided in the config file. There are three things that we need to know to be able to load the application: the path where the module resides on the disk, the name of the module, and the name of the application within the module. Using these, WSGI class first loads the module from the given path and then extracts the application from the module. When the request arrives, the process method of this class is called with the parsed request and a callable through which any data can be sent to the client. In the process method, we create the parameters according to the WSGI specification and then invoke the loaded application with those parameters.

Once the application returns, we already know the HTTP status code to respond with and all the response headers to send. We send those first, and then we iterate through the result returned by the application and send all of those as the response body.

That's the end-to-end flow of how a request arrives at the server, is passed on to the application, and then the result produced by the application is sent back as the response.

The Final Frontier

Now that the server is done, how about we write an application that can use this server? But wait, that would mean the application will have to deal with all this WSGI stuff. In our day-to-day work, that is not something that the application developer deals with. Then who does? The web framework.

Why don’t we write a simple web framework before we go on to write the application.

```python
import importlib
import os

ENVIRONMENT_VARIABLE = 'PYDEV_SETTINGS_MODULE'


class WSGIHandler:

    def __init__(self):
        settings_module = os.environ.get(ENVIRONMENT_VARIABLE)
        if not settings_module:
            raise Exception('PYDEV_SETTINGS_MODULE environment setting not set')

        mod = importlib.import_module(settings_module)
        self.url_mapping = mod.URL_CONF

    def __call__(self, *args, **kwargs):
        env = args[0]
        start_response = args[1]

        request = Request(env)
        handler = self.url_mapping.get(request.path, None)
        if not handler:
            start_response('404 Not Found', [])
            return [str.encode(f'No mapping found for url {request.path}')]
        else:
            response = handler(request)
            start_response(response.status, response.headers)
            return [str.encode(response.body)]


class Request:

    def __init__(self, env):
        self.path = env['PATH_INFO']
        self.method = env['REQUEST_METHOD']


class Response:

    def __init__(self):
        self.status = '200 OK'
        self.headers = []
        self.body = ''

    def add_header(self, name, value):
        self.headers.append((name, value))
```

```
    def set_body(self, body):
        if body:
            self.body = body


def get_wsgi_application():
    return WSGIHandler()
```

framework.py hosted with ❤ by GitHub view raw

Let's call our framework PyDev. PyDev works a lot like Django. It requires you to have a settings module, the path to which is set in an environment variable. The framework loads the module dynamically. This module defines all the stuff that's needed to set up the framework.

In our case, the only thing this module will give us is a dictionary of URL-to-function mappings so that the framework knows which function of the application to invoke when the request comes in for a particular URL. The framework also defines its own request and response classes so that the application does not need to deal with the environment dictionary provided by the server based on the WSGI specification.

The framework has a WSGIHandler that's actually the application that the server will invoke. Its `__call__` method would then proxy the request to the appropriate application function based on the request path. It understands the WSGI interface and communicates with the server accordingly, thereby abstracting the entire process from the application developer.

Let's look at an application built using this framework.

The application will have three files: one corresponding to the views (i.e., the functions invoked in response to a URL being requested), one for the settings required by the PyDev framework, and one that will be the application module passed to the server.

The views

```
import pydev


def health(request):
    response = pydev.Response()
    response.set_body("Service is up and healthy")
    return response
```

views.py hosted with ❤ by GitHub view raw

A simple view function

We have just one view function here for a health check of the application. Next, we want to bind this function to a particular URL. That will happen in the settings file.

The settings

```
from .views import *

URL_CONF = {
    '/health': health,
    '/': health
}
```

settings.py hosted with ❤️ by GitHub view raw

Settings file with URL_CONF configured

We have mapped `both / and /health` to the same view. You can imagine an application having a lot of view functions with very complex logic. For the framework, the only thing that matters is the URL and corresponding view function.

WSGI application module

```
import os

import pydev

os.environ.setdefault('PYDEV_SETTINGS_MODULE', 'pyapp.settings')

application = pydev.get_wsgi_application()
```

wsgi.py hosted with ❤️ by GitHub view raw

The WSGI application module

In this module, we set the `PYDEV_SETTINGS_MODULE` environment variable to point to the right module in our application. Then we create an attribute called `application` by calling the `get_wsgi_application` function of the PyDev framework. This function returns an instance of the WSGIHandler class. This `application` object will be used by the server to interact with our application. In Django, this file is auto-generated when `startapp` is run.

We can use the following config file to deploy this application on the server.

```yaml
address: 0.0.0.0
port: 9989
concurrency: 5
backlog: 100
timeout: 5
app-loc: /Users/vaibhav.sinha/Documents/Work/pyserve/testing/pyapp
app-module: wsgi
app: application
```

app.yaml hosted with ❤️ by GitHub view raw

With that, we have created an HTTP server that implements the WSGI specification and developed a framework that can be used to build applications that would be deployed on this server. That's it, really. That's how servers and frameworks are written. If you go look at the source code of Gunicorn, you'll find a lot of similar code. Similarly, Django has some similar code as well.

I think it should be possible to deploy this application we built on Gunicorn. Similarly, it should be possible to deploy a Django application on our server. There might be a few kinks here and there since we didn't implement the WSGI specification entirely. But this should give you a good enough idea of how things can be made interoperable by defining and implementing standard interfaces.

Epilogue

We have successfully built a Python web server from scratch. And if you look at other web servers, no matter which language they are written in, they all work the same way. They all expose some of the same configuration parameters. So now it is possible to go back to the documentation of one these famous servers and actually understand what different configurations mean, and then decide which configuration would be the best fit for our application.

Web Development | Software Development | Programming | Python | API

Follow

Written by Vaibhav Sinha

123 Followers · Writer for Better Programming

Aspiring pianist. Aspiring innovator. Aspiring entrepreneur. For now though, I write code for a living.

More from Vaibhav Sinha and Better Programming

Vaibhav Sinha in Better Programming

Write Your Own HTTP Server (Part 1)

Learn the basics of building your own HTTP server in Python

✦ · 6 min read · Jan 3, 2020

136

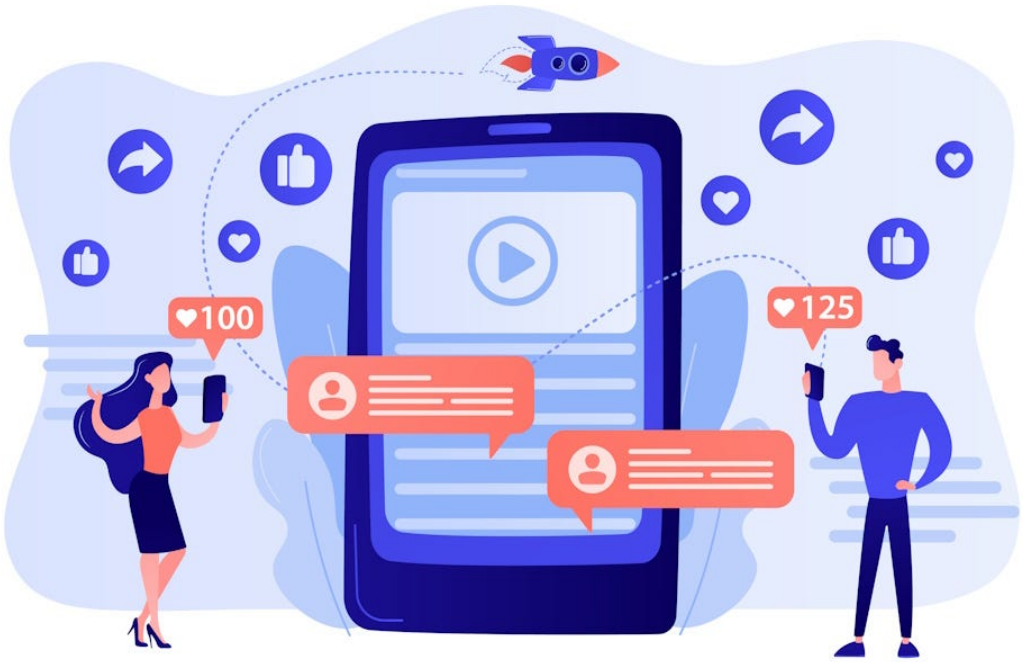
Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Practical tips for those who want to advance in their careers

22 min read · Mar 20

11.7K 229



Allen Helton in Better Programming

WebSockets, gRPC, MQTT, and SSE—Which Real-Time Notification Method Is For You?

Did you know real-time notifications were more than just WebSockets? You have plenty of options based on your use case

✦ · 7 min read · Oct 18

757 5

Vaibhav Sinha in How I Learnt Piano

One Tough Cookie

Few days back I started working on the piece Little Brown Jug from the Alfred's book. The piece seemed a bit difficult initially, mostly...

2 min read · Sep 11, 2017

See all from Vaibhav Sinha

See all from Better Programming

Recommended from Medium

Vaishnav Manoj in DataX Journal

JSON is incredibly slow: Here's What's Faster!

Unlocking the Need for Speed: Optimizing JSON Performance for Lightning-Fast Apps and Finding Alternatives to it!

16 min read · Sep 28

8.4K 111

Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

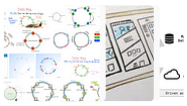
Practical tips for those who want to advance in their careers

22 min read · Mar 20

Lists

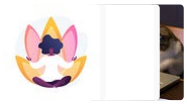
Coding & Development

11 stories · 263 saves



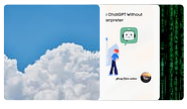
General Coding Knowledge

20 stories · 563 saves



Stories to Help You Grow as a Software Developer

19 stories · 539 saves



ChatGPT

22 stories · 252 saves

Arslan Ahmad in Level Up Coding

Don’t Just LeetCode; Follow the Coding Patterns Instead

What if you don’t like to practice 100s of coding questions before the interview?

5 min read · Sep 15, 2022

6.2K 33

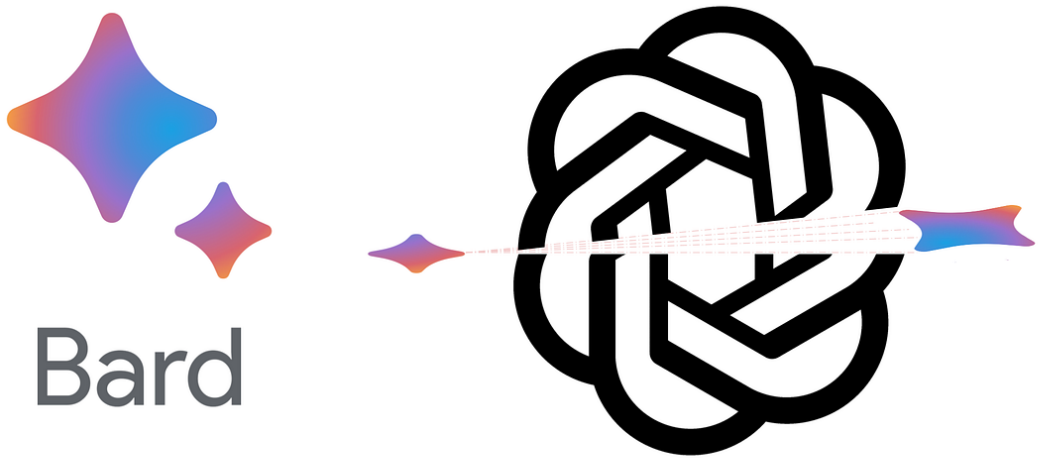
Tom Jay

Stop using Integer ID’s in your Database

I've seen this over and over for the last 30 years, people let the database set the ID or Primary Key of a table from the database, at...

✦ · 3 min read · May 22

2.2K 143



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

✦ · 6 min read · Sep 1

18.5K 563

Emmanuel Odii

Bye-bye useState & useEffect: Revolutionizing React Development!

Today, I want to show you an alternative for the useState and useEffect hook in React. (It reduces a lot of boilerplate codes)

7 min read · May 16

3.7K 75