An ASGI web server, for Python.

Test Suite passing | pypi package 0.31.0 | python 3.8 | 3.9 | 3.10 | 3.11 | 3.12

Introduction

Uvicorn is an ASGI web server implementation for Python.

Until recently Python has lacked a minimal low-level server/application interface for async frameworks. The ASGI specification fills this gap, and means we're now able to start building a common set of tooling usable across all async frameworks.

Uvicorn currently supports **HTTP/1.1** and **WebSockets**.

Quickstart

Install using `pip`:

```
$ pip install uvicorn
```

This will install uvicorn with minimal (pure Python) dependencies.

```
$ pip install 'uvicorn[standard]'
```

This will install uvicorn with "Cython-based" dependencies (where possible) and other "optional extras".

In this context, "Cython-based" means the following:

- the event loop `uvloop` will be installed and used if possible.
- `uvloop` is a fast, drop-in replacement of the built-in asyncio event loop. It is implemented in Cython. Read more here.
- The built-in asyncio event loop serves as an easy-to-read reference implementation and is there for easy debugging as it's pure-python based.
- the http protocol will be handled by `httptools` if possible.
- Read more about comparison with `h11` here.

Moreover, "optional extras" means that:

- the websocket protocol will be handled by `websockets` (should you want to use `wsproto` you'd need to install it manually) if possible.
- the `--reload` flag in development mode will use `watchfiles`.
- windows users will have `colorama` installed for the colored logs.
- `python-dotenv` will be installed should you want to use the `--env-file` option.
- `PyYAML` will be installed to allow you to provide a `.yaml` file to `--log-config`, if desired.

Create an application:

main.py

```python
async def app(scope, receive, send):
    assert scope['type'] == 'http'

    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
        ],
    })
    await send({
        'type': 'http.response.body',
        'body': b'Hello, world!',
    })
```

Run the server:

```
$ uvicorn main:app
```

Usage

The uvicorn command line tool is the easiest way to run your application.

Command line options

```
$ uvicorn --help
Usage: uvicorn [OPTIONS] APP

Options:
  --host TEXT                     Bind socket to this host.  [default:
                                  127.0.0.1]
  --port INTEGER                  Bind socket to this port. If 0, an available
                                  port will be picked.  [default: 8000]
  --uds TEXT                      Bind to a UNIX domain socket.
  --fd INTEGER                    Bind to socket from this file descriptor.
  --reload                        Enable auto-reload.
  --reload-dir PATH               Set reload directories explicitly, instead
                                  of using the current working directory.
  --reload-include TEXT           Set glob patterns to include while watching
                                  for files. Includes '*.py' by default; these
                                  defaults can be overridden with `--reload-
                                  exclude`. This option has no effect unless
                                  watchfiles is installed.
  --reload-exclude TEXT           Set glob patterns to exclude while watching
                                  for files. Includes '.*, .py[cod], .sw.*,
                                  ~*' by default; these defaults can be
                                  overridden with `--reload-include`. This
                                  option has no effect unless watchfiles is
                                  installed.
  --reload-delay FLOAT            Delay between previous and next check if
                                  application needs to be. Defaults to 0.25s.
                                  [default: 0.25]
  --workers INTEGER               Number of worker processes. Defaults to the
                                  $WEB_CONCURRENCY environment variable if
                                  available, or 1. Not valid with --reload.
  --loop [auto|asyncio|uvloop]    Event loop implementation.  [default: auto]
  --http [auto|h11|httptools]     HTTP protocol implementation.  [default:
                                  auto]
  --ws [auto|none|websockets|wsproto]
                                  WebSocket protocol implementation.
                                  [default: auto]
  --ws-max-size INTEGER           WebSocket max size message in bytes
                                  [default: 16777216]
  --ws-max-queue INTEGER          The maximum length of the WebSocket message
                                  queue.  [default: 32]
  --ws-ping-interval FLOAT        WebSocket ping interval in seconds.
                                  [default: 20.0]
  --ws-ping-timeout FLOAT         WebSocket ping timeout in seconds.
```

```
                                  [default: 20.0]
  --ws-per-message-deflate BOOLEAN
                                  WebSocket per-message-deflate compression
                                  [default: True]
  --lifespan [auto|on|off]        Lifespan implementation.  [default: auto]
  --interface [auto|asgi3|asgi2|wsgi]
                                  Select ASGI3, ASGI2, or WSGI as the
                                  application interface.  [default: auto]
  --env-file PATH                 Environment configuration file.
  --log-config PATH               Logging configuration file. Supported
                                  formats: .ini, .json, .yaml.
  --log-level [critical|error|warning|info|debug|trace]
                                  Log level. [default: info]
  --access-log / --no-access-log  Enable/Disable access log.
  --use-colors / --no-use-colors  Enable/Disable colorized logging.
  --proxy-headers / --no-proxy-headers
                                  Enable/Disable X-Forwarded-Proto,
                                  X-Forwarded-For, X-Forwarded-Port to
                                  populate remote address info.
  --server-header / --no-server-header
                                  Enable/Disable default Server header.
  --date-header / --no-date-header
                                  Enable/Disable default Date header.
  --forwarded-allow-ips TEXT      Comma separated list of IP Addresses, IP
                                  Networks, or literals (e.g. UNIX Socket
                                  path) to trust with proxy headers. Defaults
                                  to the $FORWARDED_ALLOW_IPS environment
                                  variable if available, or '127.0.0.1'. The
                                  literal '*' means trust everything.
  --root-path TEXT                Set the ASGI 'root_path' for applications
                                  submounted below a given URL path.
  --limit-concurrency INTEGER     Maximum number of concurrent connections or
                                  tasks to allow, before issuing HTTP 503
                                  responses.
  --backlog INTEGER               Maximum number of connections to hold in
                                  backlog
  --limit-max-requests INTEGER    Maximum number of requests to service before
                                  terminating the process.
  --timeout-keep-alive INTEGER    Close Keep-Alive connections if no new data
                                  is received within this timeout.  [default:
                                  5]
  --timeout-graceful-shutdown INTEGER
                                  Maximum number of seconds to wait for
                                  graceful shutdown.
  --ssl-keyfile TEXT              SSL key file
  --ssl-certfile TEXT             SSL certificate file
  --ssl-keyfile-password TEXT     SSL keyfile password
  --ssl-version INTEGER           SSL version to use (see stdlib ssl module's)
                                  [default: 17]
  --ssl-cert-reqs INTEGER         Whether client certificate is required (see
                                  stdlib ssl module's)  [default: 0]
  --ssl-ca-certs TEXT             CA certificates file
  --ssl-ciphers TEXT              Ciphers to use (see stdlib ssl module's)
                                  [default: TLSv1]
  --header TEXT                   Specify custom default HTTP response headers
                                  as a Name:Value pair
  --version                       Display the uvicorn version and exit.
  --app-dir TEXT                  Look for APP in the specified directory, by
                                  adding this to the PYTHONPATH. Defaults to
                                  the current working directory.
  --h11-max-incomplete-event-size INTEGER
                                  For h11, the maximum number of bytes to
                                  buffer of an incomplete event.
  --factory                       Treat APP as an application factory, i.e. a
                                  () -> <ASGI app> callable.
  --help                          Show this message and exit.
```

For more information, see the settings documentation.

Running programmatically

There are several ways to run uvicorn directly from your application.

`uvicorn.run`

If you're looking for a programmatic equivalent of the `uvicorn` command line interface, use `uvicorn.run()`:

main.py

```python
import uvicorn

async def app(scope, receive, send):
    ...

if __name__ == "__main__":
    uvicorn.run("main:app", port=5000, log_level="info")
```

`Config` and `Server` instances

For more control over configuration and server lifecycle, use `uvicorn.Config` and `uvicorn.Server`:

main.py

```python
import uvicorn

async def app(scope, receive, send):
    ...

if __name__ == "__main__":
    config = uvicorn.Config("main:app", port=5000, log_level="info")
    server = uvicorn.Server(config)
    server.run()
```

If you'd like to run Uvicorn from an already running async environment, use `uvicorn.Server.serve()` instead:

main.py

```python
import asyncio
import uvicorn

async def app(scope, receive, send):
    ...

async def main():
    config = uvicorn.Config("main:app", port=5000, log_level="info")
    server = uvicorn.Server(config)
    await server.serve()

if __name__ == "__main__":
    asyncio.run(main())
```

Running with Gunicorn

Warning

The `uvicorn.workers` module is deprecated and will be removed in a future release.

You should use the `uvicorn-worker` package instead.

```
python -m pip install uvicorn-worker
```

Gunicorn is a mature, fully featured server and process manager.

Uvicorn includes a Gunicorn worker class allowing you to run ASGI applications, with all of Uvicorn's performance benefits, while also giving you Gunicorn's fully-featured process management.

This allows you to increase or decrease the number of worker processes on the fly, restart worker processes gracefully, or perform server upgrades without downtime.

For production deployments we recommend using gunicorn with the uvicorn worker class.

```
gunicorn example:app -w 4 -k uvicorn.workers.UvicornWorker
```

For a PyPy compatible configuration use `uvicorn.workers.UvicornH11Worker`.

For more information, see the deployment documentation.

Application factories

The `--factory` flag allows loading the application from a factory function, rather than an application instance directly. The factory will be called with no arguments and should return an ASGI application.

main.py

```
def create_app():
    app = ...
    return app
```

```
$ uvicorn --factory main:create_app
```

The ASGI interface

Uvicorn uses the ASGI specification for interacting with an application.

The application should expose an async callable which takes three arguments:

- `scope` - A dictionary containing information about the incoming connection.
- `receive` - A channel on which to receive incoming messages from the server.
- `send` - A channel on which to send outgoing messages to the server.

Two common patterns you might use are either function-based applications:

```
async def app(scope, receive, send):
    assert scope['type'] == 'http'
    ...
```

Or instance-based applications:

```
class App:
    async def __call__(self, scope, receive, send):
        assert scope['type'] == 'http'
        ...

app = App()
```

It's good practice for applications to raise an exception on scope types that they do not handle.

The content of the `scope` argument, and the messages expected by `receive` and `send` depend on the protocol being used.

The format for HTTP messages is described in the ASGI HTTP Message format.

HTTP Scope

An incoming HTTP request might have a connection `scope` like this:

```
{
    'type': 'http',
    'scheme': 'http',
    'root_path': '',
    'server': ('127.0.0.1', 8000),
    'http_version': '1.1',
    'method': 'GET',
    'path': '/',
    'headers': [
        (b'host', b'127.0.0.1:8000'),
        (b'user-agent', b'curl/7.51.0'),
        (b'accept', b'*/*')
    ]
}
```

HTTP Messages

The instance coroutine communicates back to the server by sending messages to the `send` coroutine.

```
await send({
    'type': 'http.response.start',
    'status': 200,
    'headers': [
        [b'content-type', b'text/plain'],
    ]
})
await send({
    'type': 'http.response.body',
    'body': b'Hello, world!',
})
```

Requests & responses

Here's an example that displays the method and path used in the incoming request:

```
async def app(scope, receive, send):
    """
    Echo the method and path back in an HTTP response.
    """
    assert scope['type'] == 'http'

    body = f'Received {scope["method"]} request to {scope["path"]}'
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
        ]
    })
    await send({
        'type': 'http.response.body',
        'body': body.encode('utf-8'),
    })
```

Reading the request body

You can stream the request body without blocking the asyncio task pool, by fetching messages from the `receive` coroutine.

```
async def read_body(receive):
    """
    Read and return the entire body from an incoming ASGI message.
    """
    body = b''
    more_body = True

    while more_body:
        message = await receive()
        body += message.get('body', b'')
        more_body = message.get('more_body', False)

    return body


async def app(scope, receive, send):
    """
    Echo the request body back in an HTTP response.
    """
    body = await read_body(receive)
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            (b'content-type', b'text/plain'),
            (b'content-length', str(len(body)).encode())
        ]
    })
    await send({
        'type': 'http.response.body',
        'body': body,
    })
```

Streaming responses

You can stream responses by sending multiple `http.response.body` messages to the `send` coroutine.

```
import asyncio
```

```python
async def app(scope, receive, send):
    """
    Send a slowly streaming HTTP response back to the client.
    """
    await send({
        'type': 'http.response.start',
        'status': 200,
        'headers': [
            [b'content-type', b'text/plain'],
        ]
    })
    for chunk in [b'Hello', b', ', b'world!']:
        await send({
            'type': 'http.response.body',
            'body': chunk,
            'more_body': True
        })
        await asyncio.sleep(1)
    await send({
        'type': 'http.response.body',
        'body': b'',
    })
```

Why ASGI?

Most well established Python Web frameworks started out as WSGI-based frameworks.

WSGI applications are a single, synchronous callable that takes a request and returns a response. This doesn’t allow for long-lived connections, like you get with long-poll HTTP or WebSocket connections, which WSGI doesn't support well.

Having an async concurrency model also allows for options such as lightweight background tasks, and can be less of a limiting factor for endpoints that have long periods being blocked on network I/O such as dealing with slow HTTP requests.

Alternative ASGI servers

A strength of the ASGI protocol is that it decouples the server implementation from the application framework. This allows for an ecosystem of interoperating webservers and application frameworks.

Daphne

The first ASGI server implementation, originally developed to power Django Channels, is the Daphne webserver.

It is run widely in production, and supports HTTP/1.1, HTTP/2, and WebSockets.

Any of the example applications given here can equally well be run using `daphne` instead.

```
$ pip install daphne
$ daphne app:App
```

Hypercorn

Hypercorn was initially part of the Quart web framework, before being separated out into a standalone ASGI server.

Hypercorn supports HTTP/1.1, HTTP/2, HTTP/3 and WebSockets.

```
$ pip install hypercorn
$ hypercorn app:App
```

ASGI frameworks

You can use Uvicorn, Daphne, or Hypercorn to run any ASGI framework.

For small services you can also write ASGI applications directly.

Starlette

Starlette is a lightweight ASGI framework/toolkit.

It is ideal for building high performance asyncio services, and supports both HTTP and WebSockets.

Django Channels

The ASGI specification was originally designed for use with Django Channels.

Channels is a little different to other ASGI frameworks in that it provides an asynchronous frontend onto a threaded-framework backend. It allows Django to support WebSockets, background tasks, and long-running connections, with application code still running in a standard threaded context.

Quart

Quart is a Flask-like ASGI web framework.

FastAPI

FastAPI is an API framework based on **Starlette** and **Pydantic**, heavily inspired by previous server versions of **APIStar**.

You write your API function parameters with Python 3.6+ type declarations and get automatic data conversion, data validation, OpenAPI schemas (with JSON Schemas) and interactive API documentation UIs.

BlackSheep

BlackSheep is a web framework based on ASGI, inspired by Flask and ASP.NET Core.

Its most distinctive features are built-in support for dependency injection, automatic binding of parameters by request handler's type annotations, and automatic generation of OpenAPI documentation and Swagger UI.

Falcon

Falcon is a minimalist REST and app backend framework for Python, with a focus on reliability, correctness, and performance at scale.

Muffin

Muffin is a fast, lightweight and asynchronous ASGI web-framework for Python 3.

Litestar

Litestar is a powerful, lightweight and flexible ASGI framework.

It includes everything that's needed to build modern APIs - from data serialization and validation to websockets, ORM integration, session management, authentication and more.

Panther

Panther is a fast & friendly web framework for building async APIs with Python 3.10+.

It has built-in Document-oriented Database, Caching System, Authentication and Permission Classes, Visual API Monitoring and also supports Websocket, Throttling, Middlewares.