ASGI (Asynchronous Server Gateway Interface) Specification

Version: 3.0 (2019-03-20)

Abstract

This document proposes a standard interface between network protocol servers (particularly web servers) and Python applications, intended to allow handling of multiple common protocol styles (including HTTP, HTTP/2, and WebSocket).

This base specification is intended to fix in place the set of APIs by which these servers interact and run application code; each supported protocol (such as HTTP) has a sub-specification that outlines how to encode and decode that protocol into messages.

Rationale

The WSGI specification has worked well since it was introduced, and allowed for great flexibility in Python framework and web server choice. However, its design is irrevocably tied to the HTTP-style request/response cycle, and more and more protocols that do not follow this pattern are becoming a standard part of web programming (most notably, WebSocket).

ASGI attempts to preserve a simple application interface, while providing an abstraction that allows for data to be sent and received at any time, and from different application threads or processes.

It also takes the principle of turning protocols into Python-compatible, asynchronous-friendly sets of messages and generalises it into two parts; a standardised interface for communication around which to build servers (this document), and a set of standard message formats for each protocol.

Its primary goal is to provide a way to write HTTP/2 and WebSocket code alongside normal HTTP handling code, however; part of this design means ensuring there is an easy path to use both existing WSGI servers and applications, as a large majority of Python web [illegible] WSGI and providing an easy path forward is critical to adoption. Details on that interoperability are covered in the ASGI-HTTP spec.

Overview

ASGI consists of two different components:

- A *protocol server*, which terminates sockets and translates them into connections and per-connection event messages.
- An *application*, which lives inside a *protocol server*, is called once per connection, and handles event messages as they happen, emitting its own event messages back when necessary.

Like WSGI, the server hosts the application inside it, and dispatches incoming requests to it in a standardized format. Unlike WSGI, however, applications are asynchronous callables rather than simple callables, and they communicate with the server by receiving and sending asynchronous event messages rather than receiving a single input stream and returning a single iterable. ASGI applications must run as `async` / `await` compatible coroutines (i.e. `asyncio`-compatible) (on the main thread; they are free to use threading or other processes if they need synchronous code).

Unlike WSGI, there are two separate parts to an ASGI connection:

- A *connection scope*, which represents a protocol connection to a user and survives until the connection closes.
- *Events*, which are messages sent to the application as things happen on the connection, and messages sent back by the application to be received by the server, including data to be transmitted to the client.

Applications are called and awaited with a connection `scope` and two awaitable callables to `receive` event messages and `send` event messages back. All this happening in an asynchronous event loop.

Each call of the application callable maps to a single incoming "socket" or connection, and is expected to last the lifetime of that connection plus a little longer if there is cleanup to do. Some protocols may not use traditional sockets; ASGI specifications for those protocols are expected to define what the scope lifetime is and when it gets shut down.

Specification Details

Connection Scope

Every connection by a user to an ASGI application results in a call of the applica handle that connection entirely. How long this lives, and the information that describes each specific connection, is called the *connection scope*.

Closely related, the first argument passed to an application callable is a `scope` dictionary with all the information describing that specific connection.

For example, under HTTP the connection scope lasts just one request, but the `scope` passed contains most of the request data (apart from the HTTP request body, as this is streamed in via events).

Under WebSocket, though, the connection scope lasts for as long as the socket is connected. And the `scope` passed contains information like the WebSocket's path, but details like incoming messages come through as events instead.

Some protocols may give you a `scope` with very limited information up front because they encapsulate something like a handshake. Each protocol definition must contain information about how long its connection scope lasts, and what information you will get in the `scope` parameter.

Depending on the protocol spec, applications may have to wait for an initial opening message before communicating with the client.

Events

ASGI decomposes protocols into a series of *events* that an application must *receive* and react to, and *events* the application might *send* in response. For HTTP, this is as simple as *receiving* two events in order - `http.request` and `http.disconnect`, and *sending* the corresponding event messages back. For something like a WebSocket, it could be more like *receiving* `websocket.connect`, *sending* a `websocket.send`, *receiving* a `websocket.receive`, and finally *receiving* a `websocket.disconnect`.

Each event is a `dict` with a top-level `type` key that contains a Unicode string of the message type. Users are free to invent their own message types and send them between application instances for high-level events - for example, a chat application might send chat messages with a user type of `mychat.message`. It is expected that applications should be able to handle a mixed set of events, some sourced from the incoming client connection and some from other parts of the application.

Because these messages could be sent over a network, they need to be serializable, and so they are only allowed to contain the following types:

- Byte strings
- Unicode strings
- Integers (within the signed 64-bit range)

- Floating point numbers (within the IEEE 754 double precision range; no `Nan` or infinities)
- Lists (tuples should be encoded as lists)
- Dicts (keys must be Unicode strings)
- Booleans
- `None`

Applications

Note

The application format changed in 3.0 to use a single callable, rather than the prior two-callable format. Two-callable is documented below in "Legacy Applications"; servers can easily implement support for it using the `asgiref.compatibility` library, and should try to support it.

ASGI applications should be a single async callable:

```
coroutine application(scope, receive, send)
```

- `scope`: The connection scope information, a dictionary that contains at least a `type` key specifying the protocol that is incoming
- `receive`: an awaitable callable that will yield a new event dictionary when one is available
- `send`: an awaitable callable taking a single event dictionary as a positional argument that will return once the send has been completed or the connection has been closed

The application is called once per "connection". The definition of a connection and its lifespan are dictated by the protocol specification in question. For example, with HTTP it is one request, whereas for a WebSocket it is a single WebSocket connection.

Both the `scope` and the format of the event messages you send and receive are defined by one of the application protocols. `scope` must be a `dict`. The key `scope["type"]` will always be present, and can be used to work out which protocol is incoming. The key `scope["asgi"]` will also be present as a dictionary containing a `scope["asgi"]["version"]` key that corresponds to the ASGI version the server implements. If missing, the version should default to `"2.0"`.

There may also be a spec-specific version present as `scope["asgi"]["spec_version"]`. This allows the individual protocol specifications to make enhancements without bumping t
version.

The protocol-specific sub-specifications cover these scope and event message formats. They are equivalent to the specification for keys in the `environ` dict for WSGI.

Legacy Applications

Legacy (v2.0) ASGI applications are defined as a callable:

```
application(scope)
```

Which returns another, awaitable callable:

```
coroutine application_instance(receive, send)
```

The meanings of `scope`, `receive` and `send` are the same as in the newer single-callable application, but note that the first callable is *synchronous*.

The first callable is called when the connection is started, and then the second callable is called and awaited immediately afterwards.

This style was retired in version 3.0 as the two-callable layout was deemed unnecessary. It's now legacy, but there are applications out there written in this style, and so it's important to support them.

There is a compatibility suite available in the `asgiref.compatibility` module which allows you to both detect legacy applications and convert them to the new single-protocol style seamlessly. Servers are encouraged to support both types as of ASGI 3.0 and gradually drop support by default over time.

Protocol Specifications

These describe the standardized scope and message formats for various protocols.

The one common key across all scopes and messages is `type`, a way to indicate what type of scope or event message is being received.

In scopes, the `type` key must be a Unicode string, like `"http"` or `"websocket"`, the relevant protocol specification.

In messages, the `type` should be namespaced as `protocol.message_type`, where the `protocol` matches the scope type, and `message_type` is defined by the protocol spec. Examples of a message `type` value include `http.request` and `websocket.send`.

Note

Applications should actively reject any protocol that they do not understand with an *Exception* (of any type). Failure to do this may result in the server thinking you support a protocol you don't, which can be confusing when using with the Lifespan protocol, as the server will wait to start until you tell it.

Current protocol specifications:

- HTTP and WebSocket
- Lifespan

Middleware

It is possible to have ASGI "middleware" - code that plays the role of both server and application, taking in a `scope` and the `send`/`receive` awaitable callables, potentially modifying them, and then calling an inner application.

When middleware is modifying the `scope`, it should make a copy of the `scope` object before mutating it and passing it to the inner application, as changes may leak upstream otherwise. In particular, you should not assume that the copy of the `scope` you pass down to the application is the one that it ends up using, as there may be other middleware in the way; thus, do not keep a reference to it and try to mutate it outside of the initial ASGI app call. Your one and only chance to add to it is before you hand control to the child application.

Error Handling

If a server receives an invalid event dictionary - for example, having an unknown `type`, missing keys an event type should have, or with wrong Python types for objects (e.g. Unicode strings for HTTP headers) - it should raise an exception out of the `send` awaitable back to the application.

If an application receives an invalid event dictionary from `receive`, it should raise an exception.

In both cases, the presence of additional keys in the event dictionary should not raise an exception. This allows non-breaking upgrades to protocol specifications over ti

Servers are free to surface errors that bubble up out of application instances they are running however they wish - log to console, send to syslog, or other options - but they must terminate the application instance and its associated connection if this happens.

Note that messages received by a server after the connection has been closed are generally not considered errors unless specified by a protocol. If no error condition is specified, the `send` awaitable callable should act as a no-op.

Even if an error is raised on `send()`, it should be an error class that the server catches and ignores if it is raised out of the application, ensuring that the server does not itself error in the process.

Extensions

There are times when protocol servers may want to provide server-specific extensions outside of a core ASGI protocol specification, or when a change to a specification is being trialled before being rolled in.

For this use case, we define a common pattern for `extensions` - named additions to a protocol specification that are optional but that, if provided by the server and understood by the application, can be used to get more functionality.

This is achieved via an `extensions` entry in the `scope` dictionary, which is itself a `dict`. Extensions have a Unicode string name that is agreed upon between servers and applications.

If the server supports an extension, it should place an entry into the `extensions` dictionary under the extension's name, and the value of that entry should itself be a `dict`. Servers can provide any extra scope information that is part of the extension inside this value or, if the extension is only to indicate that the server accepts additional events via the `send` callable, it may just be an empty `dict`.

As an example, imagine a HTTP protocol server wishes to provide an extension that allows a new event to be sent back to the server that tries to flush the network send buffer all the way through the OS level. It provides an empty entry in the `extensions` dictionary to signal that it can handle the event:

```
scope = {
    "type": "http",
    "method": "GET",
    ...
    "extensions": {
        "fullflush": {},
    },
}
```

If an application sees this it then knows it can send the custom event (say, of type `http.fullflush`) via the `send` callable.

Strings and Unicode

In this document, and all sub-specifications, *byte string* refers to the `bytes` type in Python 3. *Unicode string* refers to the `str` type in Python 3.

This document will never specify just *string* - all strings are one of the two exact types.

All `dict` keys mentioned (including those for *scopes* and *events*) are Unicode strings.

Version History

- 3.0 (2019-03-04): Changed to single-callable application style
- 2.0 (2017-11-28): Initial non-channel-layer based ASGI spec

Copyright