ASGI Documentation

Release 3.0

ASGI Team

May 16, 2022

Contents

ASGI (*Asynchronous Server Gateway Interface*) is a spiritual successor to WSGI, intended to provide a standard interface between async-capable Python web servers, frameworks, and applications.

Where WSGI provided a standard for synchronous Python apps, ASGI provides one for both asynchronous and synchronous apps, with a WSGI backwards-compatibility implementation and multiple servers and application frameworks.

You can read more in the *introduction* to ASGI, look through the *specifications*, and see what *implementations* there already are or that are upcoming.

Contribution and discussion about ASGI is welcome, and mostly happens on the asgiref GitHub repository.

CHAPTER 1

Introduction

ASGI is a spiritual successor to WSGI, the long-standing Python standard for compatibility between web servers, frameworks, and applications.

WSGI succeeded in allowing much more freedom and innovation in the Python web space, and ASGI's goal is to continue this onward into the land of asynchronous Python.

1.1 What's wrong with WSGI?

You may ask "why not just upgrade WSGI"? This has been asked many times over the years, and the problem usually ends up being that WSGI's single-callable interface just isn't suitable for more involved Web protocols like WebSocket.

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.

Even if we made this callable asynchronous, it still only has a single path to provide a request, so protocols that have multiple incoming events (like receiving WebSocket frames) can't trigger this.

1.2 How does ASGI work?

ASGI is structured as a single, asynchronous callable. It takes a `scope`, which is a `dict` containing details about the specific connection, `send`, an asynchronous callable, that lets the application send event messages to the client, and `receive`, an asynchronous callable which lets the application receive event messages from the client.

This not only allows multiple incoming events and outgoing events for each application, but also allows for a background coroutine so the application can do other things (such as listening for events on an external trigger, like a Redis queue).

In its simplest form, an application can be written as an asynchronous function, like this:

```
async def application(scope, receive, send):
    event = await receive()
    ...
    await send({"type": "websocket.send", ...})
```

Every *event* that you send or receive is a Python `dict`, with a predefined format. It's these event formats that form the basis of the standard, and allow applications to be swappable between servers.

These *events* each have a defined `type` key, which can be used to infer the event's structure. Here's an example event that you might receive from `receive` with the body from a HTTP request:

```
{
    "type": "http.request",
    "body": b"Hello World",
    "more_body": False,
}
```

And here's an example of an event you might pass to `send` to send an outgoing WebSocket message:

```
{
    "type": "websocket.send",
    "text": "Hello world!",
}
```

1.3 WSGI compatibility

ASGI is also designed to be a superset of WSGI, and there's a defined way of translating between the two, allowing WSGI applications to be run inside ASGI servers through a translation wrapper (provided in the `asgiref` library). A threadpool can be used to run the synchronous WSGI applications away from the async event loop.

CHAPTER 2

Specifications

These are the specifications for ASGI. The root specification outlines how applications are structured and called, and the protocol specifications outline the events that can be sent and received for each protocol.

2.1 ASGI (Asynchronous Server Gateway Interface) Specification

Version: 3.0 (2019-03-20)

2.1.1 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.

2.1.2 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 usage relies on WSGI and providing an easy path forward is critical to adoption. Details on that interoperability are covered in the ASGI-HTTP spec.

2.1.3 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.

2.1.4 Specification Details

Connection Scope

Every connection by a user to an ASGI application results in a call of the application callable to 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 the overall ASGI 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"`, as defined in 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 time.

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 not considered errors. In this case the `send` awaitable callable should act as a no-op.

Extra Coroutines

Frameworks or applications may want to run extra coroutines in addition to the coroutine launched for each application instance. Since there is no way to parent these to the instance's coroutine in Python 3.7, applications should ensure that all coroutines launched as part of running an application are terminated either before or at the same time as the application's coroutine.

Any coroutines that continue to run outside of this window have no guarantees about their lifetime and may be killed at any time.

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.

2.1.5 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

2.1.6 Copyright

2.2 HTTP & WebSocket ASGI Message Format

Version: 2.3 (2021-02-02)

The HTTP+WebSocket ASGI sub-specification outlines how to transport HTTP/1.1, HTTP/2 and WebSocket connections within ASGI.

It is deliberately intended and designed to be a superset of the WSGI format and specifies how to translate between the two for the set of requests that are able to be handled by WSGI.

2.2.1 Spec Versions

This spec has had three versions:

- `2.0`: The first version of the spec, released with ASGI 2.0
- `2.1`: Added the `headers` key to the WebSocket Accept response
- `2.2`: Allow `None` in the second item of `server` scope value.

- `2.3`: Added the `reason` key to the WebSocket close event.

Spec versions let you understand what the server you are using understands. If a server tells you it only supports version `2.0` of this spec, then sending `headers` with a WebSocket Accept message is an error, for example.

They are separate from the HTTP version or the ASGI version.

2.2.2 HTTP

The HTTP format covers HTTP/1.0, HTTP/1.1 and HTTP/2, as the changes in HTTP/2 are largely on the transport level. A protocol server should give different scopes to different requests on the same HTTP/2 connection, and correctly multiplex the responses back to the same stream in which they came. The HTTP version is available as a string in the scope.

Multiple header fields with the same name are complex in HTTP. RFC 7230 states that for any header field that can appear multiple times, it is exactly equivalent to sending that header field only once with all the values joined by commas.

However, RFC 7230 and RFC 6265 make it clear that this rule does not apply to the various headers used by HTTP cookies (`Cookie` and `Set-Cookie`). The `Cookie` header must only be sent once by a user-agent, but the `Set-Cookie` header may appear repeatedly and cannot be joined by commas. The ASGI design decision is to transport both request and response headers as lists of 2-element `[name, value]` lists and preserve headers exactly as they were provided.

The HTTP protocol should be signified to ASGI applications with a `type` value of `http`.

HTTP Connection Scope

HTTP connections have a single-request *connection scope* - that is, your application will be called at the start of the request, and will last until the end of that specific request, even if the underlying socket is still open and serving multiple requests.

If you hold a response open for long-polling or similar, the *connection scope* will persist until the response closes from either the client or server side.

The *connection scope* information passed in `scope` contains:

- `type` (*Unicode string*) – `"http"`.
- `asgi["version"]` (*Unicode string*) – Version of the ASGI spec.
- `asgi["spec_version"]` (*Unicode string*) – Version of the ASGI HTTP spec this server understands; one of `"2.0"`, `"2.1"`, `"2.2"` or `"2.3"`. Optional; if missing assume `2.0`.
- `http_version` (*Unicode string*) – One of `"1.0"`, `"1.1"` or `"2"`.
- `method` (*Unicode string*) – The HTTP method name, uppercased.
- `scheme` (*Unicode string*) – URL scheme portion (likely `"http"` or `"https"`). Optional (but must not be empty); default is `"http"`.
- `path` (*Unicode string*) – HTTP request target excluding any query string, with percent-encoded sequences and UTF-8 byte sequences decoded into characters.
- `raw_path` (*byte string*) – The original HTTP path component unmodified from the bytes that were received by the web server. Some web server implementations may be unable to provide this. Optional; if missing defaults to `None`.
- `query_string` (*byte string*) – URL portion after the `?`, percent-encoded.

- `root_path` (*Unicode string*) – The root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional; if missing defaults to `""`.
- `headers` (*Iterable[[byte string, byte string]]*) – An iterable of `[name, value]` two-item iterables, where `name` is the header name, and `value` is the header value. Order of header values must be preserved from the original HTTP request; order of header names is not important. Duplicates are possible and must be preserved in the message as received. Header names should be lowercased, but it is not required; servers should preserve header case on a best-effort basis. Pseudo headers (present in HTTP/2 and HTTP/3) must be removed; if `:authority` is present its value must be added to the start of the iterable with `host` as the header name or replace any existing host header already present.
- `client` (*Iterable[Unicode string, int]*) – A two-item iterable of `[host, port]`, where `host` is the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer. Optional; if missing defaults to `None`.
- `server` (*Iterable[Unicode string, Optional[int]]*) – Either a two-item iterable of `[host, port]`, where `host` is the listening address for this server, and `port` is the integer listening port, or `[path, None]` where `path` is that of the unix socket. Optional; if missing defaults to `None`.

Servers are responsible for handling inbound and outbound chunked transfer encodings. A request with a `chunked` encoded body should be automatically de-chunked by the server and presented to the application as plain body bytes; a response that is given to the server with no `Content-Length` may be chunked as the server sees fit.

Request - `receive` event

Sent to the application to indicate an incoming request. Most of the request information is in the connection `scope`; the body message serves as a way to stream large incoming HTTP bodies in chunks, and as a trigger to actually run request code (as you should not trigger on a connection opening alone).

Note that if the request is being sent using `Transfer-Encoding: chunked`, the server is responsible for handling this encoding. The `http.request` messages should contain just the decoded contents of each chunk.

Keys:

- `type` (*Unicode string*) – `"http.request"`.
- `body` (*byte string*) – Body of the request. Optional; if missing defaults to `b""`. If `more_body` is set, treat as start of body and concatenate on further chunks.
- `more_body` (*bool*) – Signifies if there is additional content to come (as part of a Request message). If `True`, the consuming application should wait until it gets a chunk with this set to `False`. If `False`, the request is complete and should be processed. Optional; if missing defaults to `False`.

Response Start - `send` event

Sent by the application to start sending a response to the client. Needs to be followed by at least one response content message. The protocol server must not start sending the response to the client until it has received at least one *Response Body* event.

You may send a `Transfer-Encoding` header in this message, but the server must ignore it. Servers handle `Transfer-Encoding` themselves, and may opt to use `Transfer-Encoding: chunked` if the application presents a response that has no `Content-Length` set.

Note that this is not the same as `Content-Encoding`, which the application still controls, and which is the appropriate place to set `gzip` or other compression flags.

Keys:

- `type` (*Unicode string*) – `"http.response.start"`.
- `status` (*int*) – HTTP status code.

- `headers` (*Iterable[[byte string, byte string]]*) – An iterable of `[name, value]` two-item iterables, where `name` is the header name, and `value` is the header value. Order must be preserved in the HTTP response. Header names must be lowercased. Optional; if missing defaults to an empty list. Pseudo headers (present in HTTP/2 and HTTP/3) must not be present.

Response Body - `send` event

Continues sending a response to the client. Protocol servers must flush any data passed to them into the send buffer before returning from a send call. If `more_body` is set to `False` this will close the connection.

Keys:

- `type` (*Unicode string*) – `"http.response.body"`.
- `body` (*byte string*) – HTTP body content. Concatenated onto any previous `body` values sent in this connection scope. Optional; if missing defaults to `b""`.
- `more_body` (*bool*) – Signifies if there is additional content to come (as part of a Response Body message). If `False`, response will be taken as complete and closed, and any further messages on the channel will be ignored. Optional; if missing defaults to `False`.

Disconnect - `receive` event

Sent to the application when a HTTP connection is closed or if `receive` is called after a response has been sent. This is mainly useful for long-polling, where you may want to trigger cleanup code if the connection closes early.

Keys:

- `type` (*Unicode string*) – `"http.disconnect"`.

2.2.3 WebSocket

WebSockets share some HTTP details - they have a path and headers - but also have more state. Again, most of that state is in the `scope`, which will live as long as the socket does.

WebSocket protocol servers should handle PING/PONG messages themselves, and send PING messages as necessary to ensure the connection is alive.

WebSocket protocol servers should handle message fragmentation themselves, and deliver complete messages to the application.

The WebSocket protocol should be signified to ASGI applications with a `type` value of `websocket`.

Websocket Connection Scope

WebSocket connections' scope lives as long as the socket itself - if the application dies the socket should be closed, and vice-versa.

The *connection scope* information passed in `scope` contains initial connection metadata (mostly from the HTTP request line and headers):

- `type` (*Unicode string*) – `"websocket"`.
- `asgi["version"]` (*Unicode string*) – The version of the ASGI spec.
- `asgi["spec_version"]` (*Unicode string*) – Version of the ASGI HTTP spec this server understands; one of `"2.0"`, `"2.1"`, `"2.2"` or `"2.3"`. Optional; if missing assume `"2.0"`.

- `http_version` (*Unicode string*) – One of `"1.1"` or `"2"`. Optional; if missing default is `"1.1"`.
- `scheme` (*Unicode string*) – URL scheme portion (likely `"ws"` or `"wss"`). Optional (but must not be empty); default is `"ws"`.
- `path` (*Unicode string*) – HTTP request target excluding any query string, with percent-encoded sequences and UTF-8 byte sequences decoded into characters.
- `raw_path` (*byte string*) – The original HTTP path component unmodified from the bytes that were received by the web server. Some web server implementations may be unable to provide this. Optional; if missing defaults to `None`.
- `query_string` (*byte string*) – URL portion after the `?`. Optional; if missing or `None` default is empty string.
- `root_path` (*Unicode string*) – The root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional; if missing defaults to empty string.
- `headers` (*Iterable[[byte string, byte string]]*) – An iterable of `[name, value]` two-item iterables, where `name` is the header name and `value` is the header value. Order should be preserved from the original HTTP request; duplicates are possible and must be preserved in the message as received. Header names should be lowercased, but it is not required; servers should preserve header case on a best-effort basis. Pseudo headers (present in HTTP/2 and HTTP/3) must be removed; if `:authority` is present its value must be added to the start of the iterable with `host` as the header name or replace any existing host header already present.
- `client` (*Iterable[Unicode string, int]*) – A two-item iterable of `[host, port]`, where `host` is the remote host's IPv4 or IPv6 address, and `port` is the remote port. Optional; if missing defaults to `None`.
- `server` (*Iterable[Unicode string, Optional[int]]*) – Either a two-item iterable of `[host, port]`, where `host` is the listening address for this server, and `port` is the integer listening port, or `[path, None]` where `path` is that of the unix socket. Optional; if missing defaults to `None`.
- `subprotocols` (*Iterable[Unicode string]*) – Subprotocols the client advertised. Optional; if missing defaults to empty list.

Connect - `receive` event

Sent to the application when the client initially opens a connection and is about to finish the WebSocket handshake.

This message must be responded to with either an *Accept* message or a *Close* message before the socket will pass `websocket.receive` messages. The protocol server must send this message during the handshake phase of the WebSocket and not complete the handshake until it gets a reply, returning HTTP status code `403` if the connection is denied.

Keys:

- `type` (*Unicode string*) – `"websocket.connect"`.

Accept - `send` event

Sent by the application when it wishes to accept an incoming connection.

- `type` (*Unicode string*) – `"websocket.accept"`.
- `subprotocol` (*Unicode string*) – The subprotocol the server wishes to accept. Optional; if missing defaults to `None`.
- `headers` (*Iterable[[byte string, byte string]]*) – An iterable of `[name, value]` two-item iterables, where `name` is the header name, and `value` is the header value. Order must be preserved in the HTTP response. Header names must be lowercased. Must not include a header named `sec-websocket-protocol`; use the

`subprotocol` key instead. Optional; if missing defaults to an empty list. *Added in spec version 2.1.* Pseudo headers (present in HTTP/2 and HTTP/3) must not be present.

Receive - `receive` event

Sent to the application when a data message is received from the client.

Keys:

- `type` (*Unicode string*) – `"websocket.receive"`.
- `bytes` (*byte string*) – The message content, if it was binary mode, or `None`. Optional; if missing, it is equivalent to `None`.
- `text` (*Unicode string*) – The message content, if it was text mode, or `None`. Optional; if missing, it is equivalent to `None`.

Exactly one of `bytes` or `text` must be non-`None`. One or both keys may be present, however.

Send - `send` event

Sent by the application to send a data message to the client.

Keys:

- `type` (*Unicode string*) – `"websocket.send"`.
- **`bytes` (*byte string*) – Binary message content, or `None`.** Optional; if missing, it is equivalent to `None`.
- **`text` (*Unicode string*) – Text message content, or `None`.** Optional; if missing, it is equivalent to `None`.

Exactly one of `bytes` or `text` must be non-`None`. One or both keys may be present, however.

Disconnect - `receive` event

Sent to the application when either connection to the client is lost, either from the client closing the connection, the server closing the connection, or loss of the socket.

Keys:

- `type` (*Unicode string*) – `"websocket.disconnect"`
- `code` (*int*) – The WebSocket close code, as per the WebSocket spec.

Close - `send` event

Sent by the application to tell the server to close the connection.

If this is sent before the socket is accepted, the server must close the connection with a HTTP 403 error code (Forbidden), and not complete the WebSocket handshake; this may present on some browsers as a different WebSocket error code (such as 1006, Abnormal Closure).

If this is sent after the socket is accepted, the server must close the socket with the close code passed in the message (or 1000 if none is specified).

- `type` (*Unicode string*) – `"websocket.close"`.
- `code` (*int*) – The WebSocket close code, as per the WebSocket spec. Optional; if missing defaults to `1000`.

- `reason` (*Unicode string*) – A reason given for the closure, can be any string. Optional; if missing or `None` default is empty string.

2.2.4 WSGI Compatibility

Part of the design of the HTTP portion of this spec is to make sure it aligns well with the WSGI specification, to ensure easy adaptability between both specifications and the ability to keep using WSGI applications with ASGI servers.

WSGI applications, being synchronous, must be run in a threadpool in order to be served, but otherwise their runtime maps onto the HTTP connection scope's lifetime.

There is an almost direct mapping for the various special keys in WSGI's `environ` variable to the `http` scope:

- `REQUEST_METHOD` is the `method`
- `SCRIPT_NAME` is `root_path`
- `PATH_INFO` can be derived from `path` and `root_path`
- `QUERY_STRING` is `query_string`
- `CONTENT_TYPE` can be extracted from `headers`
- `CONTENT_LENGTH` can be extracted from `headers`
- `SERVER_NAME` and `SERVER_PORT` are in `server`
- `REMOTE_HOST`/`REMOTE_ADDR` and `REMOTE_PORT` are in `client`
- `SERVER_PROTOCOL` is encoded in `http_version`
- `wsgi.url_scheme` is `scheme`
- `wsgi.input` is a `StringIO` based around the `http.request` messages
- `wsgi.errors` is directed by the wrapper as needed

The `start_response` callable maps similarly to `http.response.start`:

- The `status` argument becomes `status`, with the reason phrase dropped.
- `response_headers` maps to `headers`

Yielding content from the WSGI application maps to sending `http.response.body` messages.

2.2.5 WSGI encoding differences

The WSGI specification (as defined in PEP 3333) specifies that all strings sent to or from the server must be of the `str` type but only contain codepoints in the ISO-8859-1 ("latin-1") range. This was due to it originally being designed for Python 2 and its different set of string types.

The ASGI HTTP and WebSocket specifications instead specify each entry of the `scope` dict as either a byte string or a Unicode string. HTTP, being an older protocol, is sometimes imperfect at specifying encoding, so some decisions of what is Unicode versus bytes may not be obvious.

- `path`: URLs can have both percent-encoded and UTF-8 encoded sections. Because decoding these is often done by the underlying server (or sometimes even proxies in the path), this is a Unicode string, fully decoded from both UTF-8 encoding and percent encodings.
- `headers`: These are byte strings of the exact byte sequences sent by the client/to be sent by the server. While modern HTTP standards say that headers should be ASCII, older ones did not and allowed a wider range of characters. Frameworks/applications should decode headers as they deem appropriate.

- `query_string`: Unlike the `path`, this is not as subject to server interference and so is presented as its raw byte string version, percent-encoded.
- `root_path`: Unicode string to match `path`.

2.2.6 Version History

- 2.0 (2017-11-28): Initial non-channel-layer based ASGI spec

2.2.7 Copyright

This document has been placed in the public domain.

2.3 Lifespan Protocol

Version: 2.0 (2019-03-20)

The Lifespan ASGI sub-specification outlines how to communicate lifespan events such as startup and shutdown within ASGI. This refers to the lifespan of the main event loop. In a multi-process environment there will be lifespan events in each process.

The lifespan messages allow for an application to initialise and shutdown in the context of a running event loop. An example of this would be creating a connection pool and subsequently closing the connection pool to release the connections.

A possible implementation of this protocol is given below:

```
async def app(scope, receive, send):
    if scope['type'] == 'lifespan':
        while True:
            message = await receive()
            if message['type'] == 'lifespan.startup':
                ... # Do some startup here!
                await send({'type': 'lifespan.startup.complete'})
            elif message['type'] == 'lifespan.shutdown':
                ... # Do some shutdown here!
                await send({'type': 'lifespan.shutdown.complete'})
                return
    else:
        pass # Handle other types
```

2.3.1 Scope

The lifespan scope exists for the duration of the event loop.

The scope information passed in `scope` contains basic metadata:

- `type` (*Unicode string*) – `"lifespan"`.
- `asgi["version"]` (*Unicode string*) – The version of the ASGI spec.
- `asgi["spec_version"]` (*Unicode string*) – The version of this spec being used. Optional; if missing defaults to `"1.0"`.

If an exception is raised when calling the application callable with a `lifespan.startup` message or a `scope` with type `lifespan`, the server must continue but not send any lifespan events.

This allows for compatibility with applications that do not support the lifespan protocol. If you want to log an error that occurs during lifespan startup and prevent the server from starting, then send back `lifespan.startup.failed` instead.

2.3.2 Startup - `receive` event

Sent to the application when the server is ready to startup and receive connections, but before it has started to do so.

Keys:

- `type` (*Unicode string*) – `"lifespan.startup"`.

2.3.3 Startup Complete - `send` event

Sent by the application when it has completed its startup. A server must wait for this message before it starts processing connections.

Keys:

- `type` (*Unicode string*) – `"lifespan.startup.complete"`.

2.3.4 Startup Failed - `send` event

Sent by the application when it has failed to complete its startup. If a server sees this it should log/print the message provided and then exit.

Keys:

- `type` (*Unicode string*) – `"lifespan.startup.failed"`.
- `message` (*Unicode string*) – Optional; if missing defaults to `""`.

2.3.5 Shutdown - `receive` event

Sent to the application when the server has stopped accepting connections and closed all active connections.

Keys:

- `type` (*Unicode string*) – `"lifespan.shutdown"`.

2.3.6 Shutdown Complete - `send` event

Sent by the application when it has completed its cleanup. A server must wait for this message before terminating.

Keys:

- `type` (*Unicode string*) – `"lifespan.shutdown.complete"`.

2.3.7 Shutdown Failed - `send` event

Sent by the application when it has failed to complete its cleanup. If a server sees this it should log/print the message provided and then terminate.

Keys:

- `type` (*Unicode string*) – `"lifespan.shutdown.failed"`.
- `message` (*Unicode string*) – Optional; if missing defaults to `""`.

2.3.8 Version History

- 2.0 (2019-03-04): Added startup.failed and shutdown.failed, clarified exception handling during startup phase.
- 1.0 (2018-09-06): Updated ASGI spec with a lifespan protocol.

2.3.9 Copyright

2.4 ASGI TLS Extension

Version: 0.2 (2020-10-02)

This specification outlines how to report TLS (or SSL) connection information in the ASGI *connection scope* object.

2.4.1 The Base Protocol

TLS is not usable on its own, it always wraps another protocol. So this specification is not designed to be usable on its own, it must be used as an extension to another ASGI specification. That other ASGI specification is referred to as the *base protocol specification*.

For HTTP-over-TLS (HTTPS), use this TLS specification and the ASGI HTTP specification. The *base protocol specification* is the ASGI HTTP specification. (See *HTTP & WebSocket ASGI Message Format*)

For WebSockets-over-TLS (wss:// protocol), use this TLS specification and the ASGI WebSockets specification. The *base protocol specification* is the ASGI WebSockets specification. (See *HTTP & WebSocket ASGI Message Format*)

If using this extension with other protocols (not HTTPS or WebSockets), note that the *base protocol specification* must define the *connection scope* in a way that ensures it covers at most one TLS connection. If not, you cannot use this extension.

2.4.2 When to use this extension

This extension must only be used for TLS connections.

For non-TLS connections, the ASGI server is forbidden from providing this extension.

An ASGI application can check for the presence of the `"tls"` extension in the `extensions` dictionary in the connection scope. If present, the server supports this extension and the connection is over TLS. If not present, either the server does not support this extension or the connection is not over TLS.

2.4.3 TLS Connection Scope

The *connection scope* information passed in `scope` contains an `"extensions"` key, which contains a dictionary of extensions. Inside that dictionary, the key `"tls"` identifies the extension specified in this document. The value will be a dictionary with the following entries:

- `server_cert` (*Unicode string or None*) – The PEM-encoded conversion of the x509 certificate sent by the server when establishing the TLS connection. Some web server implementations may be unable to provide this (e.g. if TLS is terminated by a separate proxy or load balancer); in that case this shall be `None`. Mandatory.
- `client_cert_chain` (*Iterable[Unicode string]*) – An iterable of Unicode strings, where each string is a PEM-encoded x509 certificate. The first certificate is the client certificate. Any subsequent certificates are part of the certificate chain sent by the client, with each certificate signing the preceding one. If the client did not provide a client certificate then it will be an empty iterable. Some web server implementations may be unable to provide this (e.g. if TLS is terminated by a separate proxy or load balancer); in that case this shall be an empty iterable. Optional; if missing defaults to empty iterable.
- `client_cert_name` (*Unicode string or None*) – The x509 Distinguished Name of the Subject of the client certificate, as a single string encoded as defined in RFC4514. If the client did not provide a client certificate then it will be `None`. Some web server implementations may be unable to provide this (e.g. if TLS is terminated by a separate proxy or load balancer); in that case this shall be `None`. If `client_cert_chain` is provided and non-empty then this field must be provided and must contain information that is consistent with `client_cert_chain[0]`. Note that under some setups, (e.g. where TLS is terminated by a separate proxy or load balancer and that device forwards the client certificate name to the web server), this field may be set even where `client_cert_chain` is not set. Optional; if missing defaults to `None`.
- `client_cert_error` (*Unicode string or None*) – `None` if a client certificate was provided and successfully verified, or was not provided. If a client certificate was provided but verification failed, this is a non-empty string containing an error message or error code indicating why validation failed; the details are web server specific. Most web server implementations will reject the connection if the client certificate verification failed, instead of setting this value. However, some may be configured to allow the connection anyway. This is especially useful when testing that client certificates are supported properly by the client - it allows a response containing an error message that can be presented to a human, instead of just refusing the connection. Optional; if missing defaults to `None`.
- `tls_version` (*integer or None*) – The TLS version in use. This is one of the version numbers as defined in the TLS specifications, which is an unsigned integer. Common values include `0x0303` for TLS 1.2 or `0x0304` for TLS 1.3. If TLS is not in use, set to `None`. Some web server implementations may be unable to provide this (e.g. if TLS is terminated by a separate proxy or load balancer); in that case set to `None`. Mandatory.
- `cipher_suite` (*integer or None*) – The TLS cipher suite that is being used. This is a 16-bit unsigned integer that encodes the pair of 8-bit integers specified in the relevant RFC, in network byte order. For example RFC8446 section B.4 defines that the cipher suite `TLS_AES_128_GCM_SHA256` is `{0x13, 0x01}`; that is encoded as a `cipher_suite` value of `0x1301` (equal to 4865 decimal). Some web server implementations may be unable to provide this (e.g. if TLS is terminated by a separate proxy or load balancer); in that case set to `None`. Mandatory.

2.4.4 Events

All events are as defined in the *base protocol specification*.

2.4.5 Rationale (Informative)

This section explains the choices that led to this specification.

Providing the entire TLS certificates in `client_cert_chain`, rather than a parsed subset:

- Makes it easier for web servers to implement, as they do not have to include a parser for the entirety of the x509 certificate specifications (which are huge and complicated). They just have to convert the binary DER format certificate from the wire, to the text PEM format. That is supported by many off-the-shelf libraries.
- Makes it easier for web servers to maintain, as they do not have to update their parser when new certificate fields are defined.
- Makes it easier for clients as there are plenty of existing x509 libraries available that they can use to parse the certificate; they don't need to do some special ASGI-specific thing.
- Improves interoperability as this is a simple, well-defined encoding, that clients and servers are unlikely to get wrong.
- Makes it much easier to write this specification. There is no standard documented format for a parsed certificate in Python, and we would need to write one.
- Makes it much easier to maintain this specification. There is no need to update a parsed certificate specification when new certificate fields are defined.
- Allows the client to support new certificate fields without requiring any server changes, so long as the fields are marked as "non-critical" in the certificate. (A x509 parser is allowed to ignore non-critical fields it does not understand. Critical fields that are not understood cause certificate parsing to fail).
- Allows the client to do weird and wonderful things with the raw certificate, instead of placing arbitrary limits on it.

Specifying `tls_version` as an integer, not a string or float:

- Avoids maintenance effort in this specification. If a new version of TLS is defined, then no changes are needed in this specification.
- Does not significantly affect servers. Whatever format we specified, servers would likely need a lookup table from what their TLS library reports to what this API needs. (Unless their TLS library provides access to the raw value, in which case it can be reported via this API directly).
- Does not significantly affect clients. Whatever format we specified, clients would likely need a lookup table from what this API reports to the values they support and wish to use internally.

Specifying `cipher_suite` as an integer, not a string:

- Avoids significant effort to compile a list of cipher suites in this specification. There are a huge number of existing TLS cipher suites, many of which are not widely used, even listing them all would be a huge effort.
- Avoids maintenance effort in this specification. If a new cipher suite is defined, then no changes are needed in this specification.
- Avoids dependencies on nonstandard TLS-library-specific names. E.g. the cipher names used by OpenSSL are different from the cipher names used by the RFCs.
- Does not significantly affect servers. Whatever format we specified, (unless it was a nonstandard library-specific name and the server happened to use that library), servers would likely need a lookup table from what their TLS library reports to what this API needs. (Unless their TLS library provides access to the raw value, in which case it can be reported via this API directly).
- Does not significantly affect clients. Whatever format we specified, clients would likely need a lookup table from what this API reports to the values they support and wish to use internally.
- Using a single integer, rather than a pair of integers, makes handling this value simpler and faster.

`client_cert_name` duplicates information that is also available in `client_cert_chain`. However, many ASGI applications will probably find that information is sufficient for their application - it provides a simple string that identifies the user. It is simpler to use than parsing the x509 certificate. For the server, this information is readily available.

There are theoretical interoperability problems with `client_cert_name`, since it depends on a list of object ID names that is maintained by IANA and theoretically can change. In practice, this is not a real problem, since the object IDs that are actually used in certificates have not changed in many years. So in practice it will be fine.

2.4.6 Copyright

CHAPTER 3

Extensions

The ASGI specification provides for server-specific extensions to be used outside of the core ASGI specification. This document specifies some common extensions.

3.1 Websocket Denial Response

Websocket connections start with the client sending a HTTP request containing the appropriate upgrade headers. On receipt of this request a server can choose to either upgrade the connection or respond with an HTTP response (denying the upgrade). The core ASGI specification does not allow for any control over the denial response, instead specifying that the HTTP status code `403` should be returned, whereas this extension allows an ASGI framework to control the denial response. Rather than being a core part of ASGI, this is an extension for what is considered a niche feature as most clients do not utilise the denial response.

ASGI Servers that implement this extension will provide `websocket.http.response` in the extensions part of the scope:

```
"scope": {
    ...
    "extensions": {
        "websocket.http.response": {},
    },
}
```

This will allow the ASGI Framework to send HTTP response messages after the `websocket.connect` message. These messages cannot be followed by any other websocket messages as the server should send a HTTP response and then close the connection.

The messages themselves should be `websocket.http.response.start` and `websocket.http.response.body` with a structure that matches the `http.response.start` and `http.response.body` messages defined in the HTTP part of the core ASGI specification.

3.2 HTTP/2 Server Push

HTTP/2 allows for a server to push a resource to a client by sending a push promise. ASGI servers that implement this extension will provide `http.response.push` in the extensions part of the scope:

```
"scope": {
    ...
    "extensions": {
        "http.response.push": {},
    },
}
```

An ASGI framework can initiate a server push by sending a message with the following keys. This message can be sent at any time after the *Response Start* message but before the final *Response Body* message.

Keys:

- `type` (*Unicode string*): `"http.response.push"`
- `path` (*Unicode string*): HTTP path from URL, with percent-encoded sequences and UTF-8 byte sequences decoded into characters.
- `headers` (*Iterable[[byte string, byte string]]*): An iterable of `[name, value]` two-item iterables, where `name` is the header name, and `value` is the header value. Header names must be lowercased. Pseudo headers (present in HTTP/2 and HTTP/3) must not be present.

The ASGI server should then attempt to send a server push (or push promise) to the client. If the client supports server push, the server should create a new connection to a new instance of the application and treat it as if the client had made a request.

The ASGI server should set the pseudo `:authority` header value to be the same value as the request that triggered the push promise.

3.3 Zero Copy Send

Zero Copy Send allows you to send the contents of a file descriptor to the HTTP client with zero copy (where the underlying OS directly handles the data transfer from a source file or socket without loading it into Python and writing it out again).

ASGI servers that implement this extension will provide `http.response.zerocopysend` in the extensions part of the scope:

```
"scope": {
    ...
    "extensions": {
        "http.response.zerocopysend": {},
    },
}
```

The ASGI framework can initiate a zero-copy send by sending a message with the following keys. This message can be sent at any time after the *Response Start* message but before the final *Response Body* message, and can be mixed with `http.response.body`. It can also be called multiple times in one response. Except for the characteristics of zero-copy, it should behave the same as ordinary `http.response.body`.

Keys:

- `type` (*Unicode string*): `"http.response.zerocopysend"`

- `file` (*file descriptor object*): An opened file descriptor object with an underlying OS file descriptor that can be used to call `os.sendfile`. (e.g. not BytesIO)
- `offset` (*int*): Optional. If this value exists, it will specify the offset at which sendfile starts to read data from `file`. Otherwise, it will be read from the current position of `file`.
- `count` (*int*): Optional. `count` is the number of bytes to copy between the file descriptors. If omitted, the file will be read until its end.
- `more_body` (*bool*): Signifies if there is additional content to come (as part of a Response Body message). If `False`, response will be taken as complete and closed, and any further messages on the channel will be ignored. Optional; if missing defaults to `False`.

After calling this extension to respond, the ASGI application itself should actively close the used file descriptor - ASGI servers are not responsible for closing descriptors.

3.4 TLS

See *ASGI TLS Extension*.

CHAPTER 4

Implementations

Complete or upcoming implementations of ASGI - servers, frameworks, and other useful pieces.

4.1 Servers

4.1.1 Daphne

Stable / http://github.com/django/daphne

The current ASGI reference server, written in Twisted and maintained as part of the Django Channels project. Supports HTTP/1, HTTP/2, and WebSockets.

4.1.2 Uvicorn

Stable / https://www.uvicorn.org/

A fast ASGI server based on uvloop and httptools. Supports HTTP/1 and WebSockets.

4.1.3 Hypercorn

Beta / https://pgjones.gitlab.io/hypercorn/index.html

An ASGI server based on the sans-io hyper, h11, h2, and wsproto libraries. Supports HTTP/1, HTTP/2, and WebSockets.

4.2 Application Frameworks

4.2.1 Django/Channels

Stable / http://channels.readthedocs.io

Channels is the Django project to add asynchronous support to Django and is the original driving force behind the ASGI project. Supports HTTP and WebSockets with Django integration, and any protocol with ASGI-native code.

4.2.2 FastAPI

Beta / https://github.com/tiangolo/fastapi

FastAPI is an ASGI web framework (made with Starlette) for building web APIs based on standard Python type annotations and standards like OpenAPI, JSON Schema, and OAuth2. Supports HTTP and WebSockets.

4.2.3 Quart

Beta / https://github.com/pgjones/quart

Quart is a Python ASGI web microframework. It is intended to provide the easiest way to use asyncio functionality in a web context, especially with existing Flask apps. Supports HTTP.

4.2.4 Sanic

Beta / https://sanicframework.org

Sanic is an unopinionated and flexible web application server and framework that also has the ability to operate as an ASGI compatible framework. Therefore, it can be run using any of the ASGI web servers. Supports HTTP and WebSockets.

4.2.5 Starlette

Beta / https://github.com/encode/starlette

Starlette is a minimalist ASGI library for writing against basic but powerful `Request` and `Response` classes. Supports HTTP and WebSockets.

4.2.6 rpc.py

Beta / https://github.com/abersheeran/rpc.py

An easy-to-use and powerful RPC framework. RPC server base on WSGI & ASGI, client base on `httpx`. Supports synchronous functions, asynchronous functions, synchronous generator functions, and asynchronous generator functions. Optional use of Type hint for type conversion. Optional OpenAPI document generation.

4.3 Tools

4.3.1 a2wsgi

Stable / https://github.com/abersheeran/a2wsgi

Convert WSGI application to ASGI application or ASGI application to WSGI application. Pure Python. Only depend on the standard library.