ASGI Documentation

Release 2.0

ASGI Team

Jan 17, 2019

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 double-callable - the first callable takes a `scope`, which contains details about the incoming request, and returns a second, coroutine callable. This second callable gets given `send` and `receive` awaitables that allow the coroutine to monitor incoming events and send outgoing events.

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 a class, like this:

```
class Application:

    def __init__(self, scope):
        self.scope = scope

    async def __call__(self, 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, so you know what it is to look for. Here's an example event for sending the start of a HTTP response that you might get from `receive`:

```
{
    "type": "http.response.start",
    "status": 200,
    "headers": [b"X-Header", b"Amazing Value"],
}
```

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. There's a root specification, which specifies how applications are structured and called, and then protocol specifications, that outline the events that can be sent and received for each protocol.

2.1 ASGI (Asynchronous Server Gateway Interface) Specification

Version: 2.0 (2017-11-28)

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 are becoming a standard part of web programming that do not follow this pattern (most notably, WebSocket).

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

It also take the principle of turning protocols into Python-compatible, asynchronous-friendly sets of messages and generalises it into two parts; a standardised interface for communication and to build servers around (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, and part of this design is 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 forwards 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 instantiated once per connection, and handles event messages as they happen.

Like WSGI, the server hosts the application inside it, and dispatches incoming requests to it in a standardized format. Unlike WSGI, however, applications are instantiated objects that are fed events rather than simple callables, and must run as `asyncio`-compatible coroutines (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 sent to the application as things happen on the connection.

Applications are instantiated with a connection scope, and then run in an event loop where they are expected to handle events and send data back to the client.

Each application instance 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 (instance) 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 an instance of that application being created for the connection. How long this lives, and what information it gets given upon creation, is called the *connection scope*.

For example, under HTTP the connection scope lasts just one request, but it 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. The scope contains information like the WebSocket's path, but details like incoming messages come through as Events instead.

Some protocols may give you a connection 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 inside it.

Applications **cannot** communicate with the client when they are initialized and given their connection scope; they must wait until their event loop is entered, and depending on the protocol spec, may have to wait for an initial opening message.

Events

ASGI decomposes protocols into a series of *events* that an application must react to. For HTTP, this is as simple as two events in order - `http.request` and `http.disconnect`. For something like a WebSocket, it could be more like `websocket.connect`, `websocket.send`, `websocket.receive`, `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 would 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

ASGI applications are defined as a callable:

```
application(scope)
```

- `scope`: The Connection Scope, a dictionary that contains at least a `type` key specifying the protocol that is incoming.

This first callable is called whenever a new connection comes in to the protocol server, and creates a new *instance* of the application per connection (the instance is the object that this first callable returns).

This callable is synchronous, and must not contain blocking calls (it's recommended that all it does is store the scope). If you need to do blocking work, you must do it at the start of the next callable, before you application awaits incoming events.

It must return another, awaitable callable:

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

- `receive`, an awaitable callable that will yield a new event dict when one is available
- `send`, an awaitable callable taking a single event dict as a positional argument that will return once the send has been completed or the connection has been closed

This design is perhaps more easily recognised as one of its possible implementations, as a class:

```
class Application:

    def __init__(self, scope):
```

(continues on next page)

(continued from previous page)

```
        self.scope = scope

    async def __call__(self, receive, send):
        ...
```

The application interface is specified as the more generic case of two callables to allow more flexibility for things like factory functions or type-based dispatchers.

Both the `scope` and the format of the 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"`.

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

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 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`.

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 awaitables, 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 otherwise changes may leak upstream. 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 constructor callable that gets passed `scope`.

It's notable that the part of ASGI applications that gets to modify the `scope` runs synchronously, as it's designed to be compatible with Python class constructors. If you need to put objects into the scope that require blocking/asynchronous work to resolve, then either make them awaitables themselves, or make objects that you can fill in later during the coroutine entry (remember, the objects must be modifiable; you cannot keep a reference to the scope and try to add keys later).

Error Handling

If a server receives an invalid event dict - for example, with an unknown type, missing keys a 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 into the application.

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

In both cases, presence of additional keys in the event dict should not raise an exception. This is to allow 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 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 instance are terminated either before or at the same time as the application instance'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 a `extensions` entry in the `scope` dict, 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` dict 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 dict 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 dict 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

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

2.1.6 Copyright

2.2 HTTP & WebSocket ASGI Message Format

Version: 2.0 (2017-11-28)

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 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 requests on the same HTTP/2 connection different scopes, and correctly multiplex the responses back into the same stream as they come in. 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`.

Connection Scope

HTTP connections have a single-request connection scope - that is, your applications will be instantiated at the start of the request, and destroyed at the end, even if the underlying socket is still open and serving multiple requests.

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

The connection scope contains:

- `type`: `http`
- `http_version`: Unicode string, one of `1.0`, `1.1` or `2`.
- `method`: Unicode string 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 escapes decoded and UTF-8 byte sequences decoded into characters.
- `query_string`: Byte string URL portion after the `?`, not url-decoded.
- `root_path`: Unicode string that indicates the root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional, defaults to `""`.
- `headers`: An iterable of `[name, value]` two-item iterables, where `name` is the byte string header name, and `value` is the byte string 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 must be lowercased.
- `client`: A two-item iterable of `[host, port]`, where `host` is a unicode string of the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer. Optional, defaults to `None`.
- `server`: A two-item iterable of `[host, port]`, where `host` is the listening address for this server as a unicode string, and `port` is the integer listening port. Optional, defaults to `None`.

Request

Sent 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).

Keys:

- `type`: `http.request`
- `body`: Body of the request, as a byte string. Optional, defaults to `b""`. If `more_body` is set, treat as start of body and concatenate on further chunks.
- `more_body`: Boolean value signifying 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, defaults to `False`.

Response Start

Starts 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.

Keys:

- `type`: `http.response.start`
- `status`: Integer HTTP status code.
- `headers`: A list of `[name, value]` lists, where `name` is the byte string header name, and `value` is the byte string header value. Order must be preserved in the HTTP response. Header names must be lowercased. Optional, defaults to an empty list.

Response Body

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`: `http.response.body`
- `body`: Byte string of HTTP body content. Concatenated onto any previous `body` values sent in this connection scope. Optional, defaults to `b""`.
- `more_body`: Boolean value signifying if there is additional content to come (as part of a Response Body message). If `False`, response will be taken as complete and closed off, and any further messages on the channel will be ignored. Optional, defaults to `False`.

Disconnect

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`: `http.disconnect`

2.2.2 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`.

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 scope contains the initial connection metadata (mostly from the HTTP handshake):

- `type`: `websocket`
- `http_version`: Unicode string, one of `1.1` or `2`. Optional, 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 escapes decoded and UTF-8 byte sequences decoded into characters.
- `query_string`: Byte string URL portion after the `?`. Optional, default is empty string.
- `root_path`: Byte string that indicates the root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional, defaults to empty string.
- `headers`: An iterable of `[name, value]` two-item iterables, where `name` is the header name as byte string and `value` is the header value as a byte string. Order should be preserved from the original HTTP request; duplicates are possible and must be preserved in the message as received. Header names must be lowercased.
- `client`: A two-item iterable of `[host, port]`, where `host` is a unicode string of the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer. Optional, defaults to `None`.
- `server`: A two-item iterable of `[host, port]`, where `host` is the listening address for this server as a unicode string, and `port` is the integer listening port. Optional, defaults to `None`.
- `subprotocols`: List of subprotocols the client advertised as unicode strings. Optional, defaults to empty list.

Connection

Sent 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`: `websocket.connect`

Accept

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

- `type`: `websocket.accept`
- `subprotocol`: The subprotocol the server wishes to accept, as a unicode string. Optional, defaults to `None`.

Receive

Sent when a data message is received from the client.

Keys:

- `type`: `websocket.receive`
- `bytes`: Byte string of the message content, if it was binary mode, or `None`. Optional; if missing, it is equivalent to `None`.
- `text`: Unicode string of 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

Sends a data message to the client.

Keys:

- `type`: `websocket.send`
- **`bytes`: Byte string of binary message content, or `None`.** Optional; if missing, it is equivalent to `None`.
- **`text`: Unicode string of 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.

Disconnection

Sent 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`: `websocket.disconnect`
- `code`: The WebSocket close code (integer), as per the WebSocket spec.

Close

Tells 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`: `websocket.close`
- `code`: The WebSocket close code (integer), as per the WebSocket spec. Optional, defaults to `1000`.

2.2.3 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` key
- `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.

2.2.4 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 bytestring 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 bytestrings 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 bytestring version, undecoded.
- `root_path`: Unicode to match `path`.
- 2.0 (2017-11-28): Initial non-channel-layer based ASGI spec

This document has been placed in the public domain.

2.3 Lifespan Protocol

Version: 1.0 (2018-09-06)

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

The lifespan messages allow for a 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:

```
class App:

    def __init__(self, scope):
        self.scope = scope

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

    async def startup(self):
        ...

    async def shutdown(self):
        ...
```

2.3.1 Scope

The lifespan scope exists for the duration of the event loop. The scope itself contains basic metadata,

- `type`: `lifespan`

If an exception is raised when calling the application callable with a lifespan scope the server must continue but not send any lifespan events. This allows for compatibility with applications that do not support the lifespan protocol.

2.3.2 Startup

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

Keys:

- `type`: `lifespan.startup`

2.3.3 Startup Complete

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

Keys:

- `type`: `lifespan.startup.complete`

2.3.4 Shutdown

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

Keys:

- `type`: `lifespan.shutdown`

2.3.5 Shutdown Complete

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

Keys:

- `type`: `lifespan.shutdown.complete`

Version History

- 1.0 (2018-09-06): Updated ASGI spec with a lifespan protocol.

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 user agent 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 a 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. This is an extension, rather than a core part of ASGI, as most user agents do not utilise the denial response and hence this is a niche feature.

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`: `http.response.push`
- `path`: Unicode string HTTP path from URL, with percent escapes decoded and UTF-8 byte sequences decoded into characters.
- `headers`: A list of `[name, value]` lists, where `name` is the byte string header name, and `value` is the byte string header value. Header names must be lowercased.

The ASGI server should then attempt to send a server push (or push promise) to the client. If the client accepts, 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.

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 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.

4.2.3 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.