Channels Documentation

Release 2.0a1

Andrew Godwin

Nov 16, 2017

Contents

Channels is a project that takes Django and extends its abilities beyond HTTP - to handle WebSockets, chat protocols, IoT protocols, and more.

It does this by taking the core of Django and layering a fully asynchronous layer underneath, running Django itself in a synchronous mode but handling connections and sockets asynchronously, and giving you the choice to write in either style.

To get started understanding how Channels works, read our *Introduction*, which will walk through how things work. If you're upgrading from Channels 1, take a look at *What's new in Channels 2?* to get an overview of the changes; things are substantially different.

Warning: This is incomplete documentation for an upcoming release. If you are looking for documentation for the stable release of Channels, please select "stable" in the bottom-left corner of the page.

CHAPTER 1

Projects

Channels is comprised of several packages:

- Channels, the Django integration layer
- Daphne, the HTTP and Websocket termination server
- asgiref, the base ASGI library/memory backend
- asgi_redis, the Redis channel backend
- asgi_rabbitmq, the RabbitMQ channel backend
- asgi_ipc, the POSIX IPC channel backend

This documentation covers the system as a whole; individual release notes and instructions can be found in the individual repositories.

CHAPTER 2

Topics

2.1 Introduction

Welcome to Channels! Channels changes Django to weave asynchronous code underneath and through Django's synchronous core, allowing Django projects to handle not only HTTP, but protocols that require long-running connections too - WebSockets, MQTT, chatbots, amateur radio, and more.

It does this while preserving Django's synchronous and easy-to-use nature, allowing you to choose how your write your code - synchronous in a style like Django views, fully asynchronous, or a mixture of both. On top of this, it provides integrations with Django's auth system, session system, and more, making it easier than ever to extend your HTTP-only project to other protocols.

If you haven't get installed Channels, you may want to read *Installation* first to get it installed. This introduction isn't a direct tutorial, but you should be able to use it to follow along and make changes to an existing Django project if you like.

2.1.1 Turtles All The Way Down

Channels operates on the principle of "turtles all the way down" - we have a basic idea of what a channels "application" is, and even the simplest of *consumers* (the equivalent of Django views) are an entirely valid *ASGI* application you can run by themselves.

Note: ASGI is the name for the asynchronous server specification that Channels is built on. Like WSGI, it is designed to let you choose between different servers and frameworks rather than being locked into Channels and our server Daphne.

Channels gives you the tools to write these basic *consumers* - individual pieces that might handle chat messaging, or notifications - and tie them together with URL routing, protocol detection and other handy things to make a full application.

We treat HTTP and the existing Django views as parts of a bigger whole. Traditional Django views are still there with Channels and still useable - we wrap them up in an ASGI application called `channels.http.AsgiHandler` - but

you can now also write custom HTTP long-polling handling, or WebSocket receivers, and have that code sit alongside your existing code.

Our belief is that you want the ability to use safe, synchronous techniques like Django views for most code, but have the option to drop down to a more direct, asynchronous interface for complex tasks.

2.1.2 What is a Consumer?

A consumer is the basic unit of Channels code. We call it a *consumer* as it *consumes events*, but you can think of it as its own tiny little application. When a request or new socket comes in, Channels will follow its routing table - we'll look at that in a bit - find the right consumer for that incoming connection, and start up a copy of it.

This means that, unlike Django views, consumers are long-running. They can also be short-running - after all, HTTP requests can also be served by consumers - but they're built around the idea of living for a little while.

A basic consumer looks like this:

```
class ChatConsumer(WebsocketConsumer):

    def connect(self, message):
        self.username = "Anonymous"
        self.accept()
        self.send(text="[Welcome %s!]" % self.username)

    def receive(self, text, bytes=None):
        if text.startswith("/name"):
            self.username = text[5:].strip()
            self.send(text="[set your username to %s]" % self.username)
        else:
            self.send(text=self.username + ": " + text)

    def disconnect(self, message):
        pass
```

2.2 Installation

Channels is available on PyPI - to install it, just run:

```
pip install -U channels
```

Once that's done, you should add `channels` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    ...
    'channels',
)
```

That's it! Once enabled, `channels` will integrate itself into Django and take control of the `runserver` command. See *Introduction* for more.

Note: Please be wary of any other third-party apps that require an overloaded or replacement `runserver` command. Channels provides a separate `runserver` command and may conflict with it. An example of such a conflict is with whitenoise.runserver_nostatic from whitenoise. In order to solve such issues, try moving `channels` to the top of your `INSTALLED_APPS` or remove the offending app altogether.

2.2.1 Installing the latest development version

To install the latest version of Channels, clone the repo, change to the repo, change to the repo directory, and pip install it into your current virtual environment:

```
$ git clone git@github.com:django/channels.git
$ cd channels
$ <activate your project's virtual environment>
(environment) $ pip install -e .  # the dot specifies the current repo
```

2.3 Deploying

Channels 2 (ASGI) applications deploy similarly to WSGI applications - you load them into a server, like Daphne, and you can scale the number of server processes up and down.

The one extra requirement for a Channels project is to provision a *channel layer* - a means for the different processes to talk to each other when they need to, for example, broadcast events to each other. Both steps are covered below.

2.3.1 Setting up a channel backend

The first step is to set up a channel backend.

Typically a channel backend will connect to one or more central servers that serve as the communication layer - for example, the Redis backend connects to a Redis server. All this goes into the `CHANNEL_LAYERS` setting; here's an example for a remote Redis server:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_redis.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("redis-server-name", 6379)],
        },
    },
}
```

To use the Redis backend you have to install it:

```
pip install -U asgi_redis
```

Some backends, though, don't require an extra server, like the IPC backend, which works between processes on the same machine but not over the network (it's available in the `asgi_ipc` package):

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_ipc.IPCChannelLayer",
        "CONFIG": {
```

```
            "prefix": "mysite",
        },
    },
}
```

2.3.2 Run servers

The Channels project maintains an official ASGI server, *Daphne*, but the ASGI specification is documented to allow options between a number of servers (or to let you choose frameworks other than Channels).

If you want to support WebSockets, long-poll HTTP requests and other Channels features, you'll need to run a native ASGI interface server, as the WSGI specification has no support for running these kinds of requests concurrently.

You can just keep running your "normal" Django code as a WSGI app if you like, behind something like uwsgi or gunicorn; this won't let you support WebSockets, long-polling, or background async tasks though, so you'll need to run a separate interface server to terminate those connections and configure routing in front of your interface and WSGI servers to route requests appropriately.

If you use Daphne for all traffic, it auto-negotiates between HTTP and WebSocket, so there's no need to have your WebSockets on a separate domain or path (and they'll be able to share cookies with your normal view code, which isn't possible if you separate by domain rather than path).

To run Daphne, it just needs to be supplied with an application, much like a WSGI server would need to be.

First, make sure your project has an `asgi.py` file that looks like this (it should live next to `wsgi.py` - remember to change `myproject` below!):

```
import os
import django
from channels.routing import get_default_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "myproject.settings")
django.setup()
application = get_default_application()
```

Then, you can run Daphne and supply the channel layer as the argument:

```
daphne my_project.asgi:application
```

You should place this inside either a process supervisor (systemd, supervisord) or a container orchestration system (kubernetes, nomad) to ensure that it gets restarted if needed and to allow you to scale the number of processes.

2.4 What's new in Channels 2?

Channels 1 and Channels 2 are substantially different codebases, and the upgrade is a major one. While we have attempted to keep things as familiar and backwards-compatible as possible, major architectural shifts mean you will likely need at least some code changes to upgrade.

2.4.1 Requirements

First of all, Channels 2 is *Python 3.5 and up only*.

If you are using Python 2, or a previous version of Python 3, you cannot use Channels 2 as it relies on the `asyncio` library and native Python async support. This decision was a tough one, but ultimately Channels is a library built around async functionality and so to not use these features would be foolish in the long run.

Apart from that, there are no major changed requirements, and in fact Channels 2 deploys do not need separate worker and server processes and so should be easier to manage.

2.4.2 Conceptual Changes

The fundamental layout and concepts of how Channels work have been significantly changed; you'll need to understand how and why to help in upgrading.

Channel Layers and Processes

Channels 1 terminated HTTP and WebSocket connections in a separate process to the one that ran Django code, and shuffled requests and events between them over a cross-process *channel layer*, based on Redis or similar.

This not only meant that all request data had to be re-serialized over the network, but that you needed to deploy and scale two separate sets of servers. Channels 2 changes this by running the Django code in-process via a threadpool, meaning that the network termination and application logic are combined, like WSGI.

Application Instances

Because of this, all processing for a socket happens in the same process, so ASGI applications are now instantiated once per socket and can use local variables on `self` to store information, rather than the `channel_session` storage provided before (that is now gone entirely).

The channel layer is now only used to communicate between processes for things like broadcast messaging - in particular, you can talk to other application instances in direct events, rather than having to send directly to client sockets.

This means, for example, to broadcast a chat message, you would now send a new-chat-message event to every application instance that needed it, and the application code can handle that event, serialize the message down to the socket format, and send it out (and apply things like multiplexing).

New Consumers

Because of these changes, the way consumers work has also significantly changed. Channels 2 is now a turtles-all-the-way-down design; every aspect of the system is designed as a valid ASGI application, including consumers and the routing system.

The consumer base classes have changed, though if you were using the generic consumers before, the way they work is broadly similar. However, the way that user authentication, sessions, multiplexing, and similar features work has changed.

Full Async

Channels 2 is also built on a fundamental async foundation, and all servers are actually running an asynchronous event loop and only jumping to synchronous code when you interact with the Django view system or ORM. That means that you, too, can write fully asychronous code if you wish.

It's not a requirement, but it's there if you need it. We also provide convenience methods that let you jump between synchronous and asynchronous worlds easily, with correct blocking semantics, so you can write most of a consumer in an async style and then have one method that calls the Django ORM run synchronously.

2.4.3 How to Upgrade

TODO

2.5 Channels WebSocket wrapper

Channels ships with a javascript WebSocket wrapper to help you connect to your websocket and send/receive messages.

First, you must include the javascript library in your template; if you're using Django's staticfiles, this is as easy as:

```
{% load staticfiles %}

{% static "channels/js/websocketbridge.js" %}
```

If you are using an alternative method of serving static files, the compiled source code is located at `channels/static/channels/js/websocketbridge.js` in a Channels installation. We compile the file for you each release; it's ready to serve as-is.

The library is deliberately quite low-level and generic; it's designed to be compatible with any JavaScript code or framework, so you can build more specific integration on top of it.

To process messages

```
const webSocketBridge = new channels.WebSocketBridge();
webSocketBridge.connect('/ws/');
webSocketBridge.listen(function(action, stream) {
  console.log(action, stream);
});
```

To send messages, use the *send* method

```
webSocketBridge.send({prop1: 'value1', prop2: 'value1'});
```

To demultiplex specific streams

```
webSocketBridge.connect();
webSocketBridge.listen('/ws/');
webSocketBridge.demultiplex('mystream', function(action, stream) {
  console.log(action, stream);
});
webSocketBridge.demultiplex('myotherstream', function(action, stream) {
  console.info(action, stream);
});
```

To send a message to a specific stream

```
webSocketBridge.stream('mystream').send({prop1: 'value1', prop2: 'value1'})
```

The *WebSocketBridge* instance exposes the underlaying *ReconnectingWebSocket* as the *socket* property. You can use this property to add any custom behavior. For example

```
webSocketBridge.socket.addEventListener('open', function() {
    console.log("Connected to WebSocket");
})
```

The library is also available as a npm module, under the name django-channels

2.6 ASGI (Asynchronous Server Gateway Interface) Draft Spec

Note: This is the second major revision of this specification, and is still in progress.

2.6.1 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, HTTP2, and WebSocket).

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

The set of sub-specifications is available *in the Message Formats section*.

2.6.2 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 *HTTP & WebSocket ASGI Message Format (Draft Spec)*.

2.6.3 Overview

ASGI consists of two different components:

- A *protocol server*, which terminates sockets and translates them into per-event messages
- An *application*, which lives inside a *protocol server*, is instantiated once per socket, 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 rather than simple callables, and must run as 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 connection to a user and survives until the user's 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.

2.6.4 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).

Under WebSocket, though, this 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 an event comes in and react to that.

Events and Messages

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.receive`, `websocket.receive`, `websocket.disconnect`.

Each message 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)
- 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 disctionary that contains at least a `type` key specifying the protocol that is incoming.

Which must return another, awaitable callable:

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

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

That instance is then called with a pair of awaitables:

- `receive`, an awaitable that will yield a new Event when one is available
- `send`, an awaitable that will return once the send has been completed

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

```
class Application:

    def __init__(self, scope):
        ...

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

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. The key `scope["type"]` will always be present, and can be used to work out which protocol is incoming.

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

Protocol Specfications

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 should be a simple string, like `"http"` or `"websocket"`, as defined in the 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`.

HTTP & WebSocket ASGI Message Format (Draft Spec)

Note: This is still in-progress, but is now mostly complete.

The HTTP+WebSocket ASGI sub-specification outlines how to transport HTTP/1.1, HTTP/2 and WebSocket connections over an ASGI-compatible channel layer.

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.

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 connection different reply channels, and correctly multiplex the responses back into the same stream as they come in. The HTTP version is available as a string in the request message.

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.

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 path from URL, with percent escapes decoded and UTF8 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`: A list of `[name, value]` lists, 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`: List 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`: List 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 cannot trigger on a connection opening alone).

Keys:

- `type`: `http.request`
- `body`: Body of the request, as a byte string. Optional, defaults to `""`. If `more_content` is set, treat as start of body and concatenate on further chunks.
- `more_content`: Boolean value signifying if there is additional content to come (as part of a Request Body Chunk 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.

Response

Starts, or 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.

Keys:

- `type`: `http.response`
- `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. Only allowed on the first response message.
- `content`: Byte string of HTTP body content. Concatenated onto any previous `content` values sent in this connection scope. Optional, defaults to `""`.
- `more_content`: Boolean value signifying if there is additional content to come (as part of a Response 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 when a HTTP connection is closed. This is mainly useful for long-polling, where you may want to trigger cleanup code if the connection closes early.

Keys:

- `type`: `http.disconnect`

WebSocket

WebSockets share some HTTP details - they have a path and headers - but also have more state. Path and header details are only sent in the connection message; applications that need to refer to these during later messages should store them in a cache or database.

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

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 HTTP headers):

- `type`: `websocket`
- `scheme`: Unicode string URL scheme portion (likely `ws` or `wss`). Optional (but must not be empty), default is `ws`.
- `path`: Unicode HTTP path from URL, already urldecoded.
- `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`: List of `[name, value]`, 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`: List 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`: List 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 ideally 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 frame is received from the client.

Keys:

- `type`: `websocket.receive`
- `bytes`: Byte string of frame content, if it was binary mode, or `None`.

- `text`: Unicode string of frame content, if it was text mode, or `None`.

Exactly one of `bytes` or `text` must be non-`None`.

Send

Sends a data frame to the client.

Keys:

- `type`: `websocket.send`
- `bytes`: Byte string of binary frame content, or `None`.
- `text`: Unicode string of text frame content, or `None`.

Exactly one of `bytes` or `text` must be non-`None`.

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

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

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 servers or applications with ASGI.

The adaptability works in two ways:

- WSGI to ASGI: A WSGI application can be written that transforms `environ` into a `http` scope and a `http.request` message, running the application inside a temporary async event loop. This solution would not allow full use of async to share CPU between requests, but would allow use of existing WSGI servers.
- ASGI to WSGI: An ASGI application can be written that translates the `http` scope and `http.request` message down into a WSGI `environ` and calls it inside a threadpool.

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 Response:

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

It may even be possible to map Request Body Chunks in a way that allows streaming of body data, though it would likely be easier and sufficient for many applications to simply buffer the whole body into memory before calling the WSGI application.

Delay Protocol ASGI Message Format (Draft Spec)

Protocol that allows any ASGI message to be delayed for a given number of milliseconds.

This simple protocol enables developers to schedule ASGI messages to be sent at a time in the future. It can be used in conjunction with any other channel. This allows you do simple tasks like scheduling an email to be sent later, to more complex tasks like testing latency in protocols.

Delay

Send a message to this channel to delay a message.

Channel: `asgi.delay`

Keys:

- `channel`: Unicode string specifying the final destination channel for the message after the delay.
- `delay`: Positive integer specifying the number of milliseconds to delay the message.
- `content`: Dictionary of unicode string keys for the message content. This should meet the

content specifications for the specified destination channel.

UDP ASGI Message Format (1.0)

Raw UDP is specified here as it is a datagram-based, unordered and unreliable protocol, which neatly maps to the underlying message abstraction. It is not expected that many applications would use the low-level protocol, but it may be useful for some.

While it might seem odd to have reply channels for UDP as it is a stateless protocol, replies need to come from the same server as the messages were sent to, so the reply channel here ensures that reply packets from an ASGI stack do not come from a different protocol server to the one you sent the initial packet to.

Receive

Sent when a UDP datagram is received.

Channel: `udp.receive`

Keys:

- `reply_channel`: Channel name for sending data, starts with `udp.send!`
- `data`: Byte string of UDP datagram payload.
- `client`: List 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.
- `server`: List 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`.

Send

Sent to send out a UDP datagram to a client.

Channel: `udp.send!`

Keys:

- `data`: Byte string of UDP datagram payload.

Protocol Format Guidelines

Message formats for protocols should follow these rules, unless a very good performance or implementation reason is present:

- If the protocol has low-level negotiation, keepalive or other features, handle these within the protocol server and don't expose them in ASGI events. You want to try and complete negotiation and present the application with rich information in its connection scope.
- If the protocol is datagram-based, one datagram should equal one ASGI message (unless size is an issue)

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.

Some serializers, such as `json`, cannot differentiate between byte strings and unicode strings; these should include logic to box one type as the other (for example, encoding byte strings as base64 unicode strings with a preceding special character, e.g. U+FFFF), so the distinction is always preserved even across the network.

2.6.5 Copyright

2.7 Community Projects

These projects from the community are developed on top of Channels:

- Djangobot, a bi-directional interface server for Slack.
- knocker, a generic desktop-notification system.
- Beatserver, a periodic task scheduler for django channels.
- cq, a simple distributed task system.
- Debugpannel, a django Debug Toolbar panel for channels.

If you'd like to add your project, please submit a PR with a link and brief description.

2.8 Contributing

If you're looking to contribute to Channels, then please read on - we encourage contributions both large and small, from both novice and seasoned developers.

2.8.1 What can I work on?

We're looking for help with the following areas:

- Documentation and tutorial writing
- Bugfixing and testing
- Feature polish and occasional new feature design
- Case studies and writeups

You can find what we're looking to work on in the GitHub issues list for each of the Channels sub-projects:

- Channels issues, for the Django integration and overall project efforts
- Daphne issues, for the HTTP and Websocket termination
- asgiref issues, for the base ASGI library/memory backend
- asgi_redis issues, for the Redis channel backend
- asgi_rabbitmq, for the RabbitMQ channel backend
- asgi_ipc issues, for the POSIX IPC channel backend

Issues are categorized by difficulty level:

- `exp/beginner`: Easy issues suitable for a first-time contributor.
- `exp/intermediate`: Moderate issues that need skill and a day or two to solve.
- `exp/advanced`: Difficult issues that require expertise and potentially weeks of work.

They are also classified by type:

- `documentation`: Documentation issues. Pick these if you want to help us by writing docs.
- `bug`: A bug in existing code. Usually easier for beginners as there's a defined thing to fix.
- `enhancement`: A new feature for the code; may be a bit more open-ended.

You should filter the issues list by the experience level and type of work you'd like to do, and then if you want to take something on leave a comment and assign yourself to it. If you want advice about how to take on a bug, leave a comment asking about it, or pop into the IRC channel at `#django-channels` on Freenode and we'll be happy to help.

The issues are also just a suggested list - any offer to help is welcome as long as it fits the project goals, but you should make an issue for the thing you wish to do and discuss it first if it's relatively large (but if you just found a small bug and want to fix it, sending us a pull request straight away is fine).

2.8.2 I'm a novice contributor/developer - can I help?

Of course! The issues labelled with `exp/beginner` are a perfect place to get started, as they're usually small and well defined. If you want help with one of them, pop into the IRC channel at `#django-channels` on Freenode or get in touch with Andrew directly at andrew@aeracode.org.

2.8.3 Can you pay me for my time?

Thanks to Mozilla, we have a reasonable budget to pay people for their time working on all of the above sorts of tasks and more. Generally, we'd prefer to fund larger projects (you can find these labelled as `epic-project` in the issues lists) to reduce the administrative overhead, but we're open to any proposal.

If you're interested in working on something and being paid, you'll need to draw up a short proposal and get in touch with the committee, discuss the work and your history with open-source contribution (we strongly prefer that you have a proven track record on at least a few things) and the amount you'd like to be paid.

If you're interested in working on one of these tasks, get in touch with Andrew Godwin (andrew@aeracode.org) as a first point of contact; he can help talk you through what's involved, and help judge/refine your proposal before it goes to the committee.

Tasks not on any issues list can also be proposed; Andrew can help talk about them and if they would be sensible to do.

2.9 Release Notes

2.9.1 1.0.0 Release Notes

Channels 1.0.0 brings together a number of design changes, including some breaking changes, into our first fully stable release, and also brings the databinding code out of alpha phase. It was released on 2017/01/08.

The result is a faster, easier to use, and safer Channels, including one major change that will fix almost all problems with sessions and connect/receive ordering in a way that needs no persistent storage.

It was unfortunately not possible to make all of the changes backwards compatible, though most code should not be too affected and the fixes are generally quite easy.

You **must also update Daphne** to at least 1.0.0 to have this release of Channels work correctly.

Major Features

Channels 1.0 introduces a couple of new major features.

WebSocket accept/reject flow

Rather than be immediately accepted, WebSockets now pause during the handshake while they send over a message on `websocket.connect`, and your application must either accept or reject the connection before the handshake is completed and messages can be received.

You **must** update Daphne to at least 1.0.0 to make this work correctly.

This has several advantages:

- You can now reject WebSockets before they even finish connecting, giving appropriate error codes to browsers and not letting the browser-side socket ever get into a connected state and send messages.
- Combined with Consumer Atomicity (below), it means there is no longer any need for the old "slight ordering" mode, as the connect consumer must run to completion and accept the socket before any messages can be received and forwarded onto `websocket.receive`.
- Any `send` message sent to the WebSocket will implicitly accept the connection, meaning only a limited set of `connect` consumers need changes (see Backwards Incompatible Changes below)

Consumer Atomicity

Consumers will now buffer messages you try to send until the consumer completes and then send them once it exits and the outbound part of any decorators have been run (even if an exception is raised).

This makes the flow of messages much easier to reason about - consumers can now be reasoned about as atomic blocks that run and then send messages, meaning that if you send a message to start another consumer you're guaranteed that the sending consumer has finished running by the time it's acted upon.

If you want to send messages immediately rather than at the end of the consumer, you can still do that by passing the `immediately` argument:

```
Channel("thumbnailing-tasks").send({"id": 34245}, immediately=True)
```

This should be mostly backwards compatible, and may actually fix race conditions in some apps that were pre-existing.

Databinding Group/Action Overhaul

Previously, databinding subclasses had to implement `group_names(instance, action)` to return what groups to send an instance's change to of the type `action`. This had flaws, most notably when what was actually just a modification to the instance in question changed its permission status so more clients could see it; to those clients, it should instead have been "created".

Now, Channels just calls `group_names(instance)`, and you should return what groups can see the instance at the current point in time given the instance you were passed. Channels will actually call the method before and after changes, comparing the groups you gave, and sending out create, update or delete messages to clients appropriately.

Existing databinding code will need to be adapted; see the "Backwards Incompatible Changes" section for more.

Demultiplexer Overhaul

Demuliplexers have changed to remove the behaviour where they re-sent messages onto new channels without special headers, and instead now correctly split out incoming messages into sub-messages that still look like `websocket.receive` messages, and directly dispatch these to the relevant consumer.

They also now forward all `websocket.connect` and `websocket.disconnect` messages to all of their sub-consumers, so it's much easier to compose things together from code that also works outside the context of multiplexing.

For more, read the updated /generic docs.

Delay Server

A built-in delay server, launched with *manage.py rundelay*, now ships if you wish to use it. It needs some extra initial setup and uses a database for persistance; see /delay for more information.

Minor Changes

- Serializers can now specify fields as `__all__` to auto-include all fields, and `exclude` to remove certain unwanted fields.
- `runserver` respects `FORCE_SCRIPT_NAME`
- Websockets can now be closed with a specific code by calling `close(status=4000)`
- `enforce_ordering` no longer has a `slight` mode (because of the accept flow changes), and is more efficient with session saving.
- `runserver` respects `--nothreading` and only launches one worker, takes a `--http-timeout` option if you want to override it from the default `60`,
- A new `@channel_and_http_session` decorator rehydrates the HTTP session out of the channel session if you want to access it inside receive consumers.
- Streaming responses no longer have a chance of being cached.
- `request.META['SERVER_PORT']` is now always a string.
- `http.disconnect` now has a `path` key so you can route it.
- Test client now has a `send_and_consume` method.

Backwards Incompatible Changes

Connect Consumers

If you have a custom consumer for `websocket.connect`, you must ensure that it either:

- Sends at least one message onto the `reply_channel` that generates a WebSocket frame (either `bytes` or `text` is set), either directly or via a group.
- Sends a message onto the `reply_channel` that is `{"accept": True}`, to accept a connection without sending data.
- Sends a message onto the `reply_channel` that is `{"close": True}`, to reject a connection mid-handshake.

Many consumers already do the former, but if your connect consumer does not send anything you MUST now send an accept message or the socket will remain in the handshaking phase forever and you'll never get any messages.

All built-in Channels consumers (e.g. in the generic consumers) have been upgraded to do this.

You **must** update Daphne to at least 1.0.0 to make this work correctly.

Databinding group_names

If you have databinding subclasses, you will have implemented `group_names(instance, action)`, which returns the groups to use based on the instance and action provided.

Now, instead, you must implement `group_names(instance)`, which returns the groups that can see the instance as it is presented for you; the action results will be worked out for you. For example, if you want to only show objects marked as "admin_only" to admins, and objects without it to everyone, previously you would have done:

```
def group_names(self, instance, action):
    if instance.admin_only:
        return ["admins"]
    else:
        return ["admins", "non-admins"]
```

Because you did nothing based on the `action` (and if you did, you would have got incomplete messages, hence this design change), you can just change the signature of the method like this:

```
def group_names(self, instance):
    if instance.admin_only:
        return ["admins"]
    else:
        return ["admins", "non-admins"]
```

Now, when an object is updated to have `admin_only = True`, the clients in the `non-admins` group will get a `delete` message, while those in the `admins` group will get an `update` message.

Demultiplexers

Demultiplexers have changed from using a `mapping` dict, which mapped stream names to channels, to using a `consumers` dict which maps stream names directly to consumer classes.

You will have to convert over to using direct references to consumers, change the name of the dict, and then you can remove any channel routing for the old channels that were in `mapping` from your routes.

Additionally, the Demultiplexer now forwards messages as they would look from a direct connection, meaning that where you previously got a decoded object through you will now get a correctly-formatted `websocket.receive` message through with the content as a `text` key, JSON-encoded. You will also now have to handle `websocket.connect` and `websocket.disconnect` messages.

Both of these issues can be solved using the `JsonWebsocketConsumer` generic consumer, which will decode for you and correctly separate connection and disconnection handling into their own methods.

2.9.2 1.0.1 Release Notes

Channels 1.0.1 is a minor bugfix release, released on 2017/01/09.

Changes

- WebSocket generic views now accept connections by default in their connect handler for better backwards compatibility.

Backwards Incompatible Changes

None.

2.9.3 1.0.2 Release Notes

Channels 1.0.2 is a minor bugfix release, released on 2017/01/12.

Changes

- Websockets can now be closed from anywhere using the new `WebsocketCloseException`, available as `channels.exceptions.WebsocketCloseException(code=None)`. There is also a generic `ChannelSocketException` you can base any exceptions on that, if it is caught, gets handed the current `message` in a `run` method, so you can do custom behaviours.
- Calling `Channel.send` or `Group.send` from outside a consumer context (i.e. in tests or management commands) will once again send the message immediately, rather than putting it into the consumer message buffer to be flushed when the consumer ends (which never happens)
- The base implementation of databinding now correctly only calls `group_names(instance)`, as documented.

Backwards Incompatible Changes

None.

2.9.4 1.0.3 Release Notes

Channels 1.0.3 is a minor bugfix release, released on 2017/02/01.

Changes

- Database connections are no longer force-closed after each test is run.
- Channel sessions are not re-saved if they're empty even if they're marked as modified, allowing logout to work correctly.
- WebsocketDemultiplexer now correctly does sessions for the second/third/etc. connect and disconnect handlers.
- Request reading timeouts now correctly return 408 rather than erroring out.
- The `rundelay` delay server now only polls the database once per second, and this interval is configurable with the `--sleep` option.

Backwards Incompatible Changes

None.

2.9.5 1.1.0 Release Notes

Channels 1.1.0 introduces a couple of major but backwards-compatible changes, including most notably the inclusion of a standard, framework-agnostic JavaScript library for easier integration with your site.

Major Changes

- Channels now includes a JavaScript wrapper that wraps reconnection and multiplexing for you on the client side. For more on how to use it, see the *Channels WebSocket wrapper* documentation.
- Test classes have been moved from `channels.tests` to `channels.test` to better match Django. Old imports from `channels.tests` will continue to work but will trigger a deprecation warning, and `channels.tests` will be removed completely in version 1.3.

Minor Changes & Bugfixes

- Bindings now support non-integer fields for primary keys on models.
- The `enforce_ordering` decorator no longer suffers a race condition where it would drop messages under high load.
- `runserver` no longer errors if the `staticfiles` app is not enabled in Django.

Backwards Incompatible Changes

None.

2.9.6 1.1.1 Release Notes

Channels 1.1.1 is a bugfix release that fixes a packaging issue with the JavaScript files.

Major Changes

None.

Minor Changes & Bugfixes

- The JavaScript binding introduced in 1.1.0 is now correctly packaged and included in builds.

Backwards Incompatible Changes

None.

2.9.7 1.1.2 Release Notes

Channels 1.1.2 is a bugfix release for the 1.1 series, released on April 1st, 2017.

Major Changes

None.

Minor Changes & Bugfixes

- Session name hash changed to SHA-1 to satisfy FIPS-140-2.
- *scheme* key in ASGI-HTTP messages now translates into *request.is_secure()* correctly.
- WebsocketBridge now exposes the underlying WebSocket as *.socket*.

Backwards Incompatible Changes

- When you upgrade all current channel sessions will be invalidated; you should make sure you disconnect all WebSockets during upgrade.

2.9.8 1.1.3 Release Notes

Channels 1.1.3 is a bugfix release for the 1.1 series, released on April 5th, 2017.

Major Changes

None.

Minor Changes & Bugfixes

- `enforce_ordering` now works correctly with the new-style process-specific channels
- ASGI channel layer versions are now explicitly checked for version compatability

Backwards Incompatible Changes

None.

2.9.9 1.1.4 Release Notes

Channels 1.1.4 is a bugfix release for the 1.1 series, released on June 15th, 2017.

Major Changes

None.

Minor Changes & Bugfixes

- Pending messages correctly handle retries in backlog situations
- Workers in threading mode now respond to ctrl-C and gracefully exit.
- `request.meta['QUERY_STRING']` is now correctly encoded at all times.
- Test client improvements
- `ChannelServerLiveTestCase` added, allows an equivalent of the Django `LiveTestCase`.
- Decorator added to check `Origin` headers (`allowed_hosts_only`)

- New `TEST_CONFIG` setting in `CHANNEL_LAYERS` that allows varying of the channel layer for tests (e.g. using a different Redis install)

Backwards Incompatible Changes

None.

2.9.10 1.1.5 Release Notes

Channels 1.1.5 is a packaging release for the 1.1 series, released on June 16th, 2017.

Major Changes

None.

Minor Changes & Bugfixes

- The Daphne dependency requirement was bumped to 1.3.0.

Backwards Incompatible Changes

None.

2.9.11 1.1.6 Release Notes

Channels 1.1.5 is a packaging release for the 1.1 series, released on June 28th, 2017.

Major Changes

None.

Minor Changes & Bugfixes

- The `runserver` `server_cls` override no longer fails with more modern Django versions that pass an `ipv6` parameter.

Backwards Incompatible Changes

None.