Channels Documentation

Release 2.0.2

Andrew Godwin

Feb 22, 2018

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.

If you would like complete code examples to read alongside the documentation or experiment on, the channels-examples repository contains well-commented example Channels projects.

> **Warning:** This is documentation for the **2.x series** of Channels. If you are looking for documentation for the legacy Channels 1, you can select `1.x` from the versions selector in the bottom-left corner.

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
- channels_redis, the Redis channel layer backend (optional)

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.

It also bundles this event-driven architecture with *channel layers*, a system that allows you to easily communicate between processes, and separate your project into different processes.

If you haven't yet 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 single 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. URL routing, middleware - they are all just ASGI applications.

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 Scopes and Events

Channels splits up incoming connections into two components: a *scope*, and a series of *events*.

The *scope* is a set of details about a single incoming connection - such as the path a web request was made from, or the originating IP address of a WebSocket, or the user messaging a chatbot - and persists throughout the connection.

For HTTP, the scope just lasts a single request. For WebSocket, it lasts for the lifetime of the socket (but changes if the socket closes and reconnects). For other protocols, it varies based on how the protocol's ASGI spec is written; for example, it's likely that a chatbot protocol would keep one scope open for the entirety of a user's conversation with the bot, even if the underlying chat protocol is stateless.

During the lifetime of this *scope*, a series of *events* occur. These represent user interactions - making a HTTP request, for example, or sending a WebSocket frame. Your Channels or ASGI applications will be **instantiated once per scope**, and then be fed the stream of *events* happening within that scope to decide what to do with.

An example with HTTP:

- The user makes a HTTP request.
- We open up a new `http` type scope with details of the request's path, method, headers, etc.
- We send a `http.request` event with the HTTP body content
- The Channels or ASGI application processes this and generates a `http.response` event to send back to the browser and close the connection.
- The HTTP request/response is completed and the scope is destroyed.

An example with a chatbot:

- The user sends a first message to the chatbot.
- This opens a scope containing the user's username, chosen name, and user ID.
- The application is given a `chat.received_message` event with the event text. It does not have to respond, but could send one, two or more other chat messages back as `chat.send_message` events if it wanted to.
- The user sends more messages to the chatbot and more `chat.received_message` events are generated.
- After a timeout or when the application process is restarted the scope is closed.

Within the lifetime of a scope - be that a chat, a HTTP request, a socket connection or something else - you will have one application instance handling all the events from it, and you can persist things onto the application instance as well. You can choose to write a raw ASGI application if you wish, but Channels gives you an easy-to-use abstraction over them called *consumers*.

2.1.3 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 (they live for the duration of a *scope*, as we described above).

A basic consumer looks like this:

```
class ChatConsumer(WebsocketConsumer):

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

    def receive(self, *, text_data):
        if text.startswith("/name"):
            self.username = text[5:].strip()
            self.send(text_data="[set your username to %s]" % self.username)
        else:
            self.send(text_data=self.username + ": " + text)

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

Each different protocol has different kinds of events that happen, and each type is represented by a different method. You write code that handles each event, and Channels will take care of scheduling them and running them all in parallel.

Underneath, Channels is running on a fully asynchronous event loop, and if you write code like above, it will get called in a synchronous thread. This means you can safely do blocking operations, like calling the Django ORM:

```
class LogConsumer(WebsocketConsumer):

    def connect(self, message):
        Log.objects.create(
            type="connected",
            client=self.scope["client"],
        )
```

However, if you want more control and you're willing to work only in asynchronous functions, you can write fully asynchronous consumers:

```
class PingConsumer(AsyncConsumer):
    async def websocket_connect(self, message):
        await self.send({
            "type": "websocket.accept",
        })

    async def websocket_receive(self, message):
        await asyncio.sleep(1)
        await self.send({
            "type": "websocket.send",
            "text": "pong",
        })
```

You can read more about consumers in *Consumers*.

2.1.4 Routing and Multiple Protocols

You can combine multiple Consumers (which are, remember, their own ASGI apps) into one bigger app that represents your project using routing:

```
application = URLRouter([
    url("^chat/admin/$", AdminChatConsumer),
    url("^chat/$", PublicChatConsumer),
])
```

Channels is not just built around the world of HTTP and WebSockets - it also allows you to build any protocol into a Django environment, by building a server that maps those protocols into a similar set of events. For example, you can build a chatbot in a similar style:

```
class ChattyBotConsumer(SyncConsumer):

    def telegram_message(self, message):
        """
        Simple echo handler for telegram messages in any chat.
        """
        self.send({
            "type": "telegram.message",
            "text": "You said: %s" % message["text"],
        })
```

And then use another router to have the one project able to serve both WebSockets and chat requests:

```
application = ProtocolTypeRouter({

    "websocket": URLRouter([
        url("^chat/admin/$", AdminChatConsumer),
        url("^chat/$", PublicChatConsumer),
    ]),

    "telegram": TelegramConsumer,
})
```

The goal of Channels is to let you build out your Django projects to work across any protocol or transport you might encounter in the modern web, while letting you work with the familiar components and coding style you're used to.

For more information about protocol routing, see *Routing*.

2.1.5 Cross-Process Communication

Much like a standard WSGI server, your application code that is handling protocol events runs inside the server process itself - for example, WebSocket handling code runs inside your WebSocket server process.

Each socket or connection to your overall application is handled by a *application instance* inside one of these servers. They get called and can send data back to the client directly.

However, as you build more complex application systems you start needing to communicate between different *application instances* - for example, if you are building a chatroom, when one *application instance* receives an incoming message, it needs to distribute it out to any other instances that represent people in the chatroom.

You can do this by polling a database, but Channels introduces the idea of a *channel layer*, a low-level abstraction around a set of transports that allow you to send information between different processes. Each application instance has a unique *channel name*, and can join *groups*, allowing both point-to-point and broadcast messaging.

Note: Channel layers are an optional part of Channels, and can be disabled if you want (by setting the `CHANNEL_LAYERS` setting to an empty value).

(insert cross-process example here)

You can also send messages to a dedicated process that's listening on its own, fixed channel name:

```
# In a consumer
self.channel_layer.send(
    "myproject.thumbnail_notifications",
    {
        "type": "thumbnail.generate",
        "id": 90902949,
    },
)
```

You can read more about channel layers in *Channel Layers*.

2.1.6 Django Integration

Channels ships with easy drop-in support for common Django features, like sessions and authentication. You can combine authentication with your WebSocket views by just adding the right middleware around them:

```
application = ProtocolTypeRouter({
    "websocket": AuthMiddlewareStack(
        URLRouter([
            url("^front(end)/$", consumers.AsyncChatConsumer),
        ])
    ),
})
```

For more, see *Sessions* and *Authentication*.

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',
)
```

Then, make a default routing in `myproject/routing.py`:

```
from channels.routing import ProtocolTypeRouter

application = ProtocolTypeRouter({
    # Empty for now (http->django views is added by default)
})
```

And finally, set your `ASGI_APPLICATION` setting to point to that routing object as your root application:

```
ASGI_APPLICATION = "myproject.routing.application"
```

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 Consumers

While Channels is built around a basic low-level spec called *ASGI*, it's more designed for interoperability than for writing complex applications in. So, Channels provides you with Consumers, a rich abstraction that allows you to make ASGI applications easily.

Consumers do a couple of things in particular:

- Structures your code as a series of functions to be called whenever an event happens, rather than making you write an event loop.
- Allow you to write synchronous or async code and deals with handoffs and threading for you.

Of course, you are free to ignore consumers and use the other parts of Channels - like routing, session handling and authentication - with any ASGI app, but they're generally the best way to write your application code.

2.3.1 Basic Layout

A consumer is a subclass of either `channels.consumer.AsyncConsumer` or `channels.consumer.SyncConsumer`. As these names suggest, one will expect you to write async-capable code, while the other will run your code synchronously in a threadpool for you.

Let's look at a basic example of a `SyncConsumer`:

```
from channels.consumer import SyncConsumer

class EchoConsumer(SyncConsumer):

    def websocket_connect(self, event):
        self.send({
            "type": "websocket.accept",
        })

    def websocket_receive(self, event):
        self.send({
            "type": "websocket.send",
            "text": event["text"],
        })
```

This is a very simple WebSocket echo server - it will accept all incoming WebSocket connections, and then reply to all incoming WebSocket text frames with the same text.

Consumers are structured around a series of named methods corresponding to the `type` value of the messages they are going to receive, with any `.` replaced by `_`. The two handlers above are handling `websocket.connect` and `websocket.receive` messages respectively.

How did we know what event types we were going to get and what would be in them (like `websocket.receive` having a `text`) key? That's because we designed this against the ASGI WebSocket specification, which tells us how WebSockets are presented - read more about it in *ASGI* - and protected this application with a router that checks for a scope type of `websocket` - see more about that in *Routing*.

Apart from that, the only other basic API is `self.send(event)`. This lets you send events back to the client or protocol server as defined by the protocol - if you read the WebSocket protocol, you'll see that the dict we send above is how you send a text frame to the client.

The `AsyncConsumer` is laid out very similarly, but all the handler methods must be coroutines, and `self.send` is a coroutine:

```
from channels.consumer import AsyncConsumer

class EchoConsumer(AsyncConsumer):

    async def websocket_connect(self, event):
        await self.send({
            "type": "websocket.accept",
        })

    async def websocket_receive(self, event):
        await self.send({
            "type": "websocket.send",
            "text": text,
        })
```

When should you use `AsyncConsumer` and when should you use `SyncConsumer`? The main thing to consider is what you're talking to. If you call a slow synchronous function from inside an `AsyncConsumer` you're going to hold up the entire event loop, so they're only useful if you're also calling async code (for example, using `aiohttp` to fetch 20 pages in parallel).

If you're calling any part of Django's ORM or other synchronous code, you should use a `SyncConsumer`, as this will run the whole consumer in a thread and stop your ORM queries blocking the entire server.

We recommend that you **write SyncConsumers by default**, and only use AsyncConsumers in cases where you know you are doing something that would be improved by async handling (long-running tasks that could be done in parallel)

and you are only using async-native libraries.

If you really want to call a synchronous function from an `AsyncConsumer`, take a look at `asgiref.sync.sync_to_async`, which is the utility that Channels uses to run `SyncConsumers` in threadpools, and can turn any synchronous callable into an asynchronous coroutine.

Important: If you want to call the Django ORM from an `AsyncConsumer` (or any other synchronous code), you should use the `database_sync_to_async` adapter instead. See *Database Access* for more.

Closing Consumers

When the socket or connection attached to your consumer is closed - either by you or the client - you will likely get an event sent to you (for example, `http.disconnect` or `websocket.disconnect`), and your application instance will be given a short amount of time to act on it.

Once you have finished doing your post-disconnect cleanup, you need to raise `channels.exceptions.StopConsumer` to halt the ASGI application cleanly and let the server clean it up. If you leave it running - by not raising this exception - the server will reach its application close timeout (which is 10 seconds by default in Daphne) and then kill your application and raise a warning.

The generic consumers below do this for you, so this is only needed if you are writing your own consumer class based on `AsyncConsumer` or `SyncConsumer`.

Channel Layers

Consumers also let you deal with Channel's *channel layers*, to let them send messages between each other either one-to-one or via a broadcast system called groups. You can read more in *Channel Layers*.

2.3.2 Scope

Consumers receive the connection's `scope` when they are initialised, which contains a lot of the information you'd find on the `request` object in a Django view. It's available as `self.scope` inside the consumer's methods.

Scopes are part of the *ASGI specification*, but here are some common things you might want to use:

- `scope["path"]`, the path on the request. *(HTTP and WebSocket)*
- `scope["headers"]`, raw name/value header pairs from the request *(HTTP and WebSocket)*
- `scope["method"]`, the method name used for the request. *(HTTP)*

If you enable things like *Authentication*, you'll also be able to access the user object as `scope["user"]`, and the URLRouter, for example, will put captured groups from the URL into `scope["url_route"]`.

In general, the scope is the place to get connection information and where middleware will put attributes it wants to let you access (in the same way that Django's middleware adds things to `request`).

For a full list of what can occur in a connection scope, look at the basic ASGI spec for the protocol you are terminating, plus any middleware or routing code you are using. The web (HTTP and WebSocket) scopes are available in the Web ASGI spec.

2.3.3 Generic Consumers

What you see above is the basic layout of a consumer that works for any protocol. Much like Django's *generic views*, Channels ships with *generic consumers* that wrap common functionality up so you don't need to rewrite it, specifically for HTTP and WebSocket handling.

WebsocketConsumer

Available as `channels.generic.websocket.WebsocketConsumer`, this wraps the verbose plain-ASGI message sending and receiving into handling that just deals with text and binary frames:

```
class MyConsumer(WebsocketConsumer):
    groups = ["broadcast"]

    def connect(self):
        # Called on connection. Either call
        self.accept()
        # Or to reject the connection, call
        self.close()

    def receive(self, text_data=None, bytes_data=None):
        # Called with either text_data or bytes_data for each frame
        # You can call:
        self.send(text_data="Hello world!")
        # Or, to send a binary frame:
        self.send(bytes_data="Hello world!")
        # Want to force-close the connection? Call:
        self.close()
        # Or add a custom WebSocket error code!
        self.close(code=4123)

    def disconnect(self, close_code):
        # Called when the socket closes
```

You can also raise `channels.exceptions.AcceptConnection` or `channels.exceptions.DenyConnection` from anywhere inside the `connect` method in order to accept or reject a connection, if you want reuseable authentication or rate-limiting code that doesn't need to use mixins.

A `WebsocketConsumer`'s channel will automatically be added to (on connect) and removed from (on disconnect) any groups whose names appear in the consumer's `groups` class attribute. `groups` must be an iterable, and a channel layer with support for groups must be set as the channel backend (`channels.layers.InMemoryChannelLayer` and `channels_redis.core.RedisChannelLayer` both support groups). If no channel layer is configured or the channel layer doesn't support groups, connecting to a `WebsocketConsumer` with a non-empty `groups` attribute will raise `channels.exceptions.InvalidChannelLayerError`. See *Groups* for more.

AsyncWebsocketConsumer

Available as `channels.generic.websocket.AsyncWebsocketConsumer`, this has the exact same methods and signature as `WebsocketConsumer` but everything is async, and the functions you need to write have to be as well:

```
class MyConsumer(AsyncWebsocketConsumer):
    groups = ["broadcast"]
```

(continues on next page)

(continued from previous page)

```
    async def connect(self):
        # Called on connection. Either call
        await self.accept()
        # Or to reject the connection, call
        await self.close()

    async def receive(self, text_data=None, bytes_data=None):
        # Called with either text_data or bytes_data for each frame
        # You can call:
        await self.send(text_data="Hello world!")
        # Or, to send a binary frame:
        await self.send(bytes_data="Hello world!")
        # Want to force-close the connection? Call:
        await self.close()
        # Or add a custom WebSocket error code!
        await self.close(code=4123)

    async def disconnect(self, close_code):
        # Called when the socket closes
```

JsonWebsocketConsumer

Available as `channels.generic.websocket.JsonWebsocketConsumer`, this works like `WebsocketConsumer`, except it will auto-encode and decode to JSON sent as WebSocket text frames.

The only API differences are:

- Your `receive_json` method must take a single argument, `content`, that is the decoded JSON object.
- `self.send_json` takes only a single argument, `content`, which will be encoded to JSON for you.

If you want to customise the JSON encoding and decoding, you can override the `encode_json` and `decode_json` classmethods.

AsyncJsonWebsocketConsumer

An async version of `JsonWebsocketConsumer`, available as `channels.generic.websocket.AsyncJsonWebsocketConsumer`. Note that even `encode_json` and `decode_json` are async functions.

2.4 Routing

While consumers are valid *ASGI* applications, you don't want to just write one and have that be the only thing you can give to protocol servers like Daphne. Channels provides routing classes that allow you to combine and stack your consumers (and any other valid ASGI application) to dispatch based on what the connection is.

Important: Channels routers only work on the *scope* level, not on the level of individual *events*, which means you can only have one consumer for any given connection. Routing is to work out what single consumer to give a connection, not how to spread events from one connection across multiple consumers.

Routers are themselves valid ASGI applications, and it's possible to nest them. We suggest that you have a `ProtocolTypeRouter` as the root application of your project - the one that you pass to protocol servers - and nest other, more protocol-specific routing underneath there.

Channels expects you to be able to define a single *root application*, and provide the path to it as the `ASGI_APPLICATION` setting (think of this as being analagous to the `ROOT_URLCONF` setting in Django). There's no fixed rule as to where you need to put the routing and the root application, but we recommend putting them in a project-level file called `routing.py`, next to `urls.py`. You can read more about deploying Channels projects and settings in *Deploying*.

Here's an example of what that `routing.py` might look like:

```
from django.conf.urls import url

from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack

from chat.consumers import AdminChatConsumer, PublicChatConsumer
from aprs_news.consumers import APRSNewsConsumer

application = ProtocolTypeRouter({

    # WebSocket chat handler
    "websocket": AuthMiddlewareStack(
        URLRouter([
            url("^chat/admin/$", AdminChatConsumer),
            url("^chat/$", PublicChatConsumer),
        ])
    ),

    # Using the third-party project frequensgi, which provides an APRS protocol
    "aprs": APRSNewsConsumer,

})
```

It's possible to have routers from third-party apps, too, or write your own, but we'll go over the built-in Channels ones here.

2.4.1 ProtocolTypeRouter

`channels.routing.ProtocolTypeRouter`

This should be the top level of your ASGI application stack and the main entry in your routing file.

It lets you dispatch to one of a number of other ASGI applications based on the `type` value present in the `scope`. Protocols will define a fixed type value that their scope contains, so you can use this to distinguish between incoming connection types.

It takes a single argument - a dictionary mapping type names to ASGI applications that serve them:

```
ProtocolTypeRouter({
    "http": some_app,
    "websocket": some_other_app,
})
```

If a `http` argument is not provided, it will default to the Django view system's ASGI interface, `channels.http.AsgiHandler`, which means that for most projects that aren't doing custom long-poll HTTP handling, you can simply not specify a `http` option and leave it to work the "normal" Django way.

If you want to split HTTP handling between long-poll handlers and Django views, use a URLRouter with `channels.http.AsgiHandler` specified as the last entry with a match-everything pattern.

2.4.2 URLRouter

`channels.routing.URLRouter`

Routes `http` or `websocket` type connections via their HTTP path. Takes a single argument, a list of Django URL objects (either `path()` or `url()`):

```
URLRouter([
    url("^longpoll/$", LongPollConsumer),
    url("^notifications/(?P<stream>\w+)/$", LongPollConsumer),
    url("^$", AsgiHandler),
])
```

Any captured groups will be provided in `scope` as the key `url_route`, a dict with an `args` key containing a list of positional regex groups and a `kwargs` key with a dict of the named regex groups.

For example, to pull out the named group `stream` in the example above, you would do this:

```
stream = self.scope["url_route"]["kwargs"]["stream"]
```

2.4.3 ChannelNameRouter

`channels.routing.ChannelNameRouter`

Routes `channel` type scopes based on the value of the `channel` key in their scope. Intended for use with the *Worker and Background Tasks*.

It takes a single argument - a dictionary mapping channel names to ASGI applications that serve them:

```
ChannelNameRouter({
    "thumbnails-generate": some_app,
    "thunbnails-delete": some_other_app,
})
```

2.5 Database Access

The Django ORM is a synchronous piece of code, and so if you want to access it from asynchronous code you need to do special handling to make sure its connections are closed properly.

If you're using `SyncConsumer`, or anything based on it - like `JsonWebsocketConsumer` - you don't need to do anything special, as all your code is already run in a synchronous mode and Channels will do the cleanup for you as part of the `SyncConsumer` code.

If you are writing asynchronous code, however, you will need to call database methods in a safe, synchronous context, using `database_sync_to_async`.

2.5.1 database_sync_to_async

`channels.db.database_sync_to_async` is a version of `asgiref.sync.sync_to_async` that also cleans up database connections on exit.

To use it, write your ORM queries in a separate function or method, and then call it with `database_sync_to_async` like so:

```
from channels.db import database_sync_to_async

async def connect(self):
    self.username = await database_sync_to_async(self.get_name)()

def get_name(self):
    return User.objects.all()[0].name
```

You can also use it as a decorator:

```
from channels.db import database_sync_to_async

async def connect(self):
    self.username = await self.get_name()

@database_sync_to_async
def get_name(self):
    return User.objects.all()[0].name
```

2.6 Channel Layers

Channel layers allow you to talk between different instances of an application. They're a useful part of making a distributed realtime application if you don't want to have to shuttle all of your messages or events through a database.

Additionally, they can also be used in combination with a worker process to make a basic task queue or to offload tasks - read more in *Worker and Background Tasks*.

Channels does not ship with any channel layers you can use out of the box, as each one depends on a different way of transporting data across a network. We would recommend you use `channels_redis`, which is an offical Django-maintained layer that uses Redis as a transport and what we'll focus the examples on here.

Note: Channel layers are an entirely optional part of Channels as of version 2.0. If you don't want to use them, just leave `CHANNEL_LAYERS` unset, or set it to the empty dict `{}`.

Messages across channel layers also go to consumers/ASGI application instances, just like events from the client, and so they now need a `type` key as well. See more below.

Warning: Channel layers have a purely async interface (for both send and receive); you will need to wrap them in a converter if you want to call them from synchronous code (see below).

2.6.1 Configuration

Channel layers are configured via the `CHANNEL_LAYERS` Django setting. It generally looks something like this:

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

(continues on next page)

(continued from previous page)

```
    },
}
```

You can get the default channel layer from a project with `channels.layers.get_channel_layer()`, but if you are using consumers a copy is automatically provided for you on the consumer as `self.channel_layer`.

2.6.2 Synchronous Functions

By default the `send()`, `group_send()`, `group_add()` and other functions are async functions, meaning you have to `await` them. If you need to call them from synchronous code, you'll need to use the handy `asgiref.sync.async_to_sync` wrapper:

```
from asgiref.sync import async_to_sync

async_to_sync(channel_layer.send)("channel_name", {...})
```

2.6.3 What To Send Over The Channel Layer

Unlike in Channels 1, the channel layer is only for high-level application-to-application communication. When you send a message, it is received by the consumers listening to the group or channel on the other end, and not transported to that consumer's socket directly.

What this means is that you should send high-level events over the channel layer, and then have consumers handle those events and do appropriate low-level networking to their attached client.

For example, the multichat example in Andrew Godwin's `channels-examples` repository sends events like this over the channel layer:

```
await self.channel_layer.group_send(
    room.group_name,
    {
        "type": "chat.message",
        "room_id": room_id,
        "username": self.scope["user"].username,
        "message": message,
    }
)
```

And then the consumers define a handling function to receive those events and turn them into WebSocket frames:

```
async def chat_message(self, event):
    """
    Called when someone has messaged our chat.
    """
    # Send a message down to the client
    await self.send_json(
        {
            "msg_type": settings.MSG_TYPE_MESSAGE,
            "room": event["room_id"],
            "username": event["username"],
            "message": event["message"],
        },
    )
```

Any consumer based on Channels' `SyncConsumer` or `AsyncConsumer` will automatically provide you a `self.channel_layer` and `self.channel_name` attribute, which contains a pointer to the channel layer instance and the channel name that will reach the consumer respectively.

Any message sent to that channel name - or to a group the channel name was added to - will be received by the consumer much like an event from its connected client, and dispatched to a named method on the consumer. The name of the method will be the `type` of the event with periods replaced by underscores - so, for example, an event coming in over the channel layer with a `type` of `chat.join` will be handled by the method `chat_join`.

Note: If you are inheriting from the `AsyncConsumer` class tree, all your event handlers, including ones for events over the channel layer, must be asynchronous (`async def`). If you are in the `SyncConsumer` class tree instead, they must all be synchronous (`def`).

2.6.4 Single Channels

Each application instance - so, for example, each long-running HTTP request or open WebSocket - results in a single Consumer instance, and if you have channel layers enabled, Consumers will generate a unique *channel name* for themselves, and start listening on it for events.

This means you can send those consumers events from outside the process - from other consumers, maybe, or from management commands - and they will react to them and run code just like they would events from their client connection.

The channel name is available on a consumer as `self.channel_name`. Here's an example of writing the channel name into a database upon connection, and then specifying a handler method for events on it:

```
class ChatConsumer(WebsocketConsumer):

    def connect(self):
        # Make a database row with our channel name
        Clients.objects.create(channel_name=self.channel_name)

    def disconnect(self, close_code):
        # Note that in some rare cases (power loss, etc) disconnect may fail
        # to run; this naive example would leave zombie channel names around.
        Clients.objects.filter(channel_name=self.channel_name).delete()

    def chat_message(self, event):
        # Handles the "chat.message" event when it's sent to us.
        self.send(text_data=event["text"])
```

Note that, because you're mixing event handling from the channel layer and from the protocol connection, you need to make sure that your type names do not clash. It's recommended you prefix type names (like we did here with `chat.`) to avoid clashes.

To send to a single channel, just find its channel name (for the example above, we could crawl the database), and use `channel_layer.send`:

```
from channels.layers import get_channel_layer

channel_layer = get_channel_layer()
await channel_layer.send("channel_name", {
    "type": "chat.message",
    "text": "Hello there!",
})
```

2.6.5 Groups

Obviously, sending to individual channels isn't particularly useful - in most cases you'll want to send to multiple channels/consumers at once as a broadcast. Not only for cases like a chat where you want to send incoming messages to everyone in the room, but even for sending to an individual user who might have more than one browser tab or device connected.

You can construct your own solution for this if you like, using your existing datastores, or use the Groups system built-in to some channel layers. Groups are a broadcast system that:

- Allows you to add and remove channel names from named groups, and send to those named groups.
- Provides group expiry for clean-up of connections whose disconnect handler didn't get to run (e.g. power failure)

They do not allow you to enumerate or list the channels in a group; it's a pure broadcast system. If you need more precise control or need to know who is connected, you should build your own system or use a suitable third-party one.

You use groups by adding a channel to them during connection, and removing it during disconnection, illustrated here on the WebSocket generic consumer:

```
# This example uses WebSocket consumer, which is synchronous, and so
# needs the async channel layer functions to be converted.
from asgiref.sync import async_to_sync

class ChatConsumer(WebsocketConsumer):

    def connect(self):
        async_to_sync(self.channel_layer.group_add)("chat", self.channel_name)

    def disconnect(self, close_code):
        async_to_sync(self.channel_layer.group_discard)("chat", self.channel_name)
```

Then, to send to a group, use `group_send`, like in this small example which broadcasts chat messages to every connected socket when combined with the code above:

```
class ChatConsumer(WebsocketConsumer):

    ...

    def receive(self, text_data):
        async_to_sync(self.channel_layer.group_send)(
            "chat",
            {
                "type": "chat.message",
                "text": text_data,
            },
        )

    def chat_message(self, event):
        self.send(text_data=event["text"])
```

2.6.6 Using Outside Of Consumers

You'll often want to send to the channel layer from outside of the scope of a consumer, and so you won't have `self.channel_layer`. In this case, you should use the `get_channel_layer` function to retrieve it:

```
from channels.layers import get_channel_layer
channel_layer = get_channel_layer()
```

Then, once you have it, you can just call methods on it. Remember that **channel layers only support async methods**, so you can either call it from your own asynchronous context:

```
for chat_name in chats:
    await channel_layer.group_send(
        chat_name,
        {"type": "chat.system_message", "text": announcement_text},
    )
```

Or you'll need to use async_to_sync:

```
from asgiref.sync import async_to_sync

async_to_sync(channel_layer.group_send)("chat", {"type": "chat.force_disconnect"})
```

2.7 Sessions

Channels supports standard Django sessions using HTTP cookies for both HTTP and WebSocket. There are some caveats, however.

2.7.1 Basic Usage

The `SessionMiddleware` in Channels supports standard Django sessions, and like all middleware, should be wrapped around the ASGI application that needs the session information in its scope (for example, a `URLRouter` to apply it to a whole collection of consumers, or an individual consumer).

`SessionMiddleware` requires `CookieMiddleware` to function. For convenience, these are also provided as a combined callable called `SessionMiddlewareStack` that includes both. All are importable from `channels.session`.

To use the middleware, wrap it around the appropriate level of consumer in your `routing.py`:

```
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.sessions import SessionMiddlewareStack

from myapp import consumers

application = ProtocolTypeRouter({

    "websocket": SessionMiddlewareStack(
        URLRouter([
            url("^front(end)/$", consumers.AsyncChatConsumer),
        ])
    ),

})
```

`SessionMiddleware` will only work on protocols that provide HTTP headers in their `scope` - by default, this is HTTP and WebSocket.

To access the session, use `self.scope["session"]` in your consumer code:

```
class ChatConsumer(WebsocketConsumer):

    def connect(self, event):
        self.scope["session"]["seed"] = random.randint(1, 1000)
```

`SessionMiddleware` respects all the same Django settings as the default Django session framework, like SESSION_COOKIE_NAME and SESSION_COOKIE_DOMAIN.

2.7.2 Session Persistence

Within HTTP consumers or ASGI applications, session persistence works as you would expect from Django HTTP views - sessions are saved whenever you send a HTTP response that does not have status code `500`.

This is done by overriding any `http.response.start` messages to inject cookie headers into the response as you send it out. If you have set the `SESSION_SAVE_EVERY_REQUEST` setting to `True`, it will save the session and send the cookie on every response, otherwise it will only save whenever the session is modified.

If you are in a WebSocket consumer, however, the session is populated **but will never be saved automatically** - you must call `scope["session"].save()` yourself whenever you want to persist a session to your session store. If you don't save, the session will still work correctly inside the consumer (as it's stored as an instance variable), but other connections or HTTP views won't be able to see the changes.

Note: If you are in a long-polling HTTP consumer, you might want to save changes to the session before you send a response. If you want to do this, call `scope["session"].save()`.

2.8 Authentication

Channels supports standard Django authentication out-of-the-box for HTTP and WebSocket consumers, and you can write your own middleware or handling code if you want to support a different authentication scheme (for example, tokens in the URL).

2.8.1 Django authentication

The `AuthMiddleware` in Channels supports standard Django authentication, where the user details are stored in the session. It allows read-only access to a user object in the `scope`.

`AuthMiddleware` requires `SessionMiddleware` to function, which itself requires `CookieMiddleware`. For convenience, these are also provided as a combined callable called `AuthMiddlewareStack` that includes all three.

To use the middleware, wrap it around the appropriate level of consumer in your `routing.py`:

```
from django.conf.urls import url

from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack

from myapp import consumers

application = ProtocolTypeRouter({

    "websocket": AuthMiddlewareStack(
```

(continues on next page)

(continued from previous page)

```
        URLRouter([
            url("^front(end)/$", consumers.AsyncChatConsumer),
        ])
    ),

})
```

While you can wrap the middleware around each consumer individually, it's recommended you wrap it around a higher-level application component, like in this case the `URLRouter`.

Note that the `AuthMiddleware` will only work on protocols that provide HTTP headers in their `scope` - by default, this is HTTP and WebSocket.

To access the user, just use `self.scope["user"]` in your consumer code:

```
class ChatConsumer(WebsocketConsumer):

    def connect(self, event):
        self.user = self.scope["user"]
```

2.8.2 Custom Authentication

If you have a custom authentication scheme, you can write a custom middleware to parse the details and put a user object (or whatever other object you need) into your scope.

Middleware is written as a callable that takes an ASGI application and wraps it to return another ASGI application. Most authentication can just be done on the scope, so all you need to do is override the initial constructor that takes a scope, rather than the event-running coroutine.

Here's a simple example of a middleware that just takes a user ID out of the query string and uses that:

```
class QueryAuthMiddleware:
    """
    Custom middleware (insecure) that takes user IDs from the query string.
    """

    def __init__(self, inner):
        # Store the ASGI application we were passed
        self.inner = inner

    def __call__(self, scope):
        # Look up user from query string (you should also do things like
        # check it's a valid user ID, or if scope["user"] is already populated)
        scope["user"] = User.objects.get(id=int(scope["query_string"]))
        # Return the inner application directly and let it run everything else
        return self.inner(scope)
```

The same principles can be applied to authenticate over non-HTTP protocols; for example, you might want to use someone's chat username from a chat protocol to turn it into a user.

2.9 Security

This covers basic security for protocols you're serving via Channels and helpers that we provide.

2.9.1 WebSockets

WebSockets start out life as a HTTP request, including all the cookies and headers, and so you can use the standard *Authentication* code in order to grab current sessions and check user IDs.

There is also a risk of cross-site request forgery (CSRF) with WebSockets though, as they can be initiated from any site on the internet to your domain, and will still have the user's cookies and session from your site. If you serve private data down the socket, you should restrict the sites which are allowed to open sockets to you.

This is done via the `channels.security.websocket` package, and the two ASGI middlewares it contains, `OriginValidator` and `AllowedHostsOriginValidator`.

`OriginValidator` lets you restrict the valid options for the `Origin` header that is sent with every WebSocket to say where it comes from. Just wrap it around your WebSocket application code like this, and pass it a list of valid domains as the second argument:

```
from channels.security.websocket import OriginValidator

application = ProtocolTypeRouter({

    "websocket": OriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                ...
            ])
        ),
        ["goodsite.com", "*.goodsite.com"],
    ),
})
```

Often, the set of domains you want to restrict to is the same as the Django `ALLOWED_HOSTS` setting, which performs a similar security check for the `Host` header, and so `AllowedHostsOriginValidator` lets you use this setting without having to re-declare the list:

```
from channels.security.websocket import AllowedHostsOriginValidator

application = ProtocolTypeRouter({

    "websocket": AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                ...
            ])
        ),
    ),
})
```

`AllowedHostsOriginValidator` will also automatically allow local connections through if the site is in `DEBUG` mode, much like Django's host validation.

2.10 Testing

Testing Channels consumers is a little trickier than testing normal Django views due to their underlying asynchronous nature.

To help with testing, Channels provides test helpers called *Communicators*, which allow you to wrap up an ASGI application (like a consumer) into its own event loop and ask it questions.

They do, however, require that you have asynchronous support in your test suite. While you can do this yourself, we recommend using `py.test` with its `asyncio` plugin, which is how we'll illustrate tests below.

2.10.1 Setting Up Async Tests

Firstly, you need to get `py.test` set up with async test support, and presumably Django test support as well. You can do this by installing the `pytest-django` and `pytest-asyncio` packages:

```
pip install -U pytest-django pytest-asyncio
```

Then, you need to decorate the tests you want to run async with `pytest.mark.asyncio`. Note that you can't mix this with `unittest.TestCase` subclasses; you have to write async tests as top-level test functions in the native `py.test` style:

```
import pytest
from channels.testing import HttpCommunicator
from myproject.myapp.consumers import MyConsumer

@pytest.mark.asyncio
async def test_my_consumer():
    communicator = HttpCommunicator(MyConsumer, "GET", "/test/")
    response = await communicator.get_response()
    assert response["body"] == b"test response"
    assert response["status"] == 200
```

If you have normal Django views, you can continue to test those with the standard Django test tools and client. You only need the async setup for code that's written as consumers.

There's a few variants of the Communicator - a plain one for generic usage, and one each for HTTP and WebSockets specifically that have shortcut methods,

2.10.2 ApplicationCommunicator

`ApplicationCommunicator` is the generic test helper for any ASGI application. It gives you two basic methods - one to send events and one to receive events.

Note: `ApplicationCommunicator` is actually provided by the base `asgiref` package, but we let you import it from `channels.testing` for convenience.

To construct it, pass it an application and a scope:

```
from channels.testing import ApplicationCommunicator
communicator = ApplicationCommunicator(MyConsumer, {"type": "http", ...})
```

To send an event, call `send_input`:

```
await communicator.send_input({
    "type": "http.request",
    "body": b"chunk one \x01 chunk two",
})
```

To receive an event, call `receive_output`:

```
event = await communicator.receive_output(timeout=1)
assert event["type"] == "http.response.start"
```

To wait for an application to exit (you'll need to either do this or wait for it to send you output before you can see what it did using mocks or inspection), use `wait`:

```
await communicator.wait(timeout=1)
```

If you're expecting your application to raise an exception, use `pytest.raises` around `wait`:

```
with pytest.raises(ValueError):
    await communicator.wait()
```

You should only need this generic class for non-HTTP/WebSocket tests, though you might need to fall back to it if you are testing things like HTTP chunked responses or long-polling, which aren't supported in `HttpCommunicator` yet.

2.10.3 HttpCommunicator

`HttpCommunicator` is a subclass of `ApplicationCommunicator` specifically tailored for HTTP requests. You need only instantiate it with your desired options:

```
from channels.testing import HttpCommunicator
communicator = HttpCommunicator(MyHttpConsumer, "GET", "/test/")
```

And then wait for its response:

```
response = await communicator.get_response()
assert response["body"] == b"test response"
```

You can pass the following arguments to the constructor:

- `method`: HTTP method name (unicode string, required)
- `path`: HTTP path (unicode string, required)
- `body`: HTTP body (bytestring, optional)

The response from the `get_response` method will be a dict with the following keys:

```
* ``status``: HTTP status code (integer)
* ``headers``: List of headers as (name, value) tuples (both bytestrings)
* ``body``: HTTP response body (bytestring)
```

2.10.4 WebsocketCommunicator

`WebsocketCommunicator` allows you to more easily test WebSocket consumers. It provides several convenience methods for interacting with a WebSocket application, as shown in this example:

```
from channels.testing import WebsocketCommunicator
communicator = WebsocketCommunicator(SimpleWebsocketApp, "/testws/")
connected, subprotocol = await communicator.connect()
assert connected
# Test sending text
await communicator.send_to(text_data="hello")
```

(continues on next page)

(continued from previous page)

```
response = await communicator.receive_from()
assert response == "hello"
# Close
await communicator.disconnect()
```

Note: All of these methods are coroutines, which means you must `await` them. If you do not, your test will either time out (if you forgot to await a send) or try comparing things to a coroutine object (if you forgot to await a receive).

Important: If you don't call `WebsocketCommunicator.disconnect()` before your test suite ends, you may find yourself getting `RuntimeWarnings` about things never being awaited, as you will be killing your app off in the middle of its lifecycle.

connect

Triggers the connection phase of the WebSocket and waits for the application to either accept or deny the connection. Takes no parameters and returns either:

- `(True, <chosen_subprotocol>)` if the socket was accepted. `chosen_subprotocol` defaults to `None`.
- `(False, <close_code>)` if the socket was rejected. `close_code` defaults to `1000`.

send_to

Sends a data frame to the application. Takes exactly one of `bytes_data` or `text_data` as parameters, and returns nothing:

```
await communicator.send_to(bytes_data=b"hi\0")
```

This method will type-check your parameters for you to ensure what you are sending really is text or bytes.

send_json_to

Sends a JSON payload to the application as a text frame. Call it with an object and it will JSON-encode it for you, and return nothing:

```
await communicator.send_json_to({"hello": "world"})
```

receive_from

Receives a frame from the application and gives you either `bytes` or `text` back depending on the frame type:

```
response = await communicator.receive_from()
```

Takes an optional `timeout` argument with a number of seconds to wait before timing out, which defaults to 1. It will typecheck your application's responses for you as well, to ensure that text frames contain text data, and binary frames contain binary data.

receive_json_from

Receives a text frame from the application and decodes it for you:

```
response = await communicator.receive_json_from()
assert response == {"hello": "world"}
```

Takes an optional `timeout` argument with a number of seconds to wait before timing out, which defaults to 1.

disconnect

Closes the socket from the client side. Takes nothing and returns nothing.

2.10.5 ChannelsLiveServerTestCase

If you just want to run standard Selenium or other tests that require a webserver to be running for external programs, you can use `ChannelsLiveServerTestCase`, which is a drop-in replacement for the standard Django `LiveServerTestCase`:

```
from channels.testing import ChannelsLiveServerTestCase

class SomeLiveTests(ChannelsLiveServerTestCase):

    def test_live_stuff(self):
        call_external_testing_thing(self.live_server_url)
```

serve_static

Subclass `ChannelsLiveServerTestCase` with `serve_static = True` in order to serve static files (comparable to Django's `StaticLiveServerTestCase`, you don't need to run collectstatic before or as a part of your tests setup).

2.11 Worker and Background Tasks

While *channel layers* are primarily designed for communicating between different instances of ASGI applications, they can also be used to offload work to a set of worker servers listening on fixed channel names, as a simple, very-low-latency task queue.

Note: The worker/background tasks system in Channels is simple and very fast, and achieves this by not having some features you may find useful, such as retries or return values.

We recommend you use it for work that does not need guarantees around being complete (at-most-once delivery), and for work that needs more guarantees, look into a separate dedicated task queue like Celery.

Setting up background tasks works in two parts - sending the events, and then setting up the consumers to receive and process the events.

2.11.1 Sending

To send an event, just send it to a fixed channel name. For example, let's say we want a background process that pre-caches thumbnails:

```
# Inside a consumer
self.channel_layer.send(
    "thumbnails-generate",
    {
        "type": "generate",
        "id": 123456789,
    },
)
```

Note that the event you send **must** have a `type` key, even if only one type of message is being sent over the channel, as it will turn into an event a consumer has to handle.

2.11.2 Receiving and Consumers

Channels will present incoming worker tasks to you as events inside a scope with a `type` of `channel`, and a `channel` key matching the channel name. We recommend you use ProtocolTypeRouter and ChannelNameRouter (see *Routing* for more) to arrange your consumers:

```
application = ProtocolTypeRouter({
    ...
    "channel": ChannelNameRouter({
        "thumbnails-generate": consumers.GenerateConsumer,
        "thunbnails-delete": consumers.DeleteConsumer,
    }),
})
```

You'll be specifying the `type` values of the individual events yourself when you send them, so decide what your names are going to be and write consumers to match. For example, here's a basic consumer that expects to receive an event with `type` `test.print`, and a `text` value containing the text to print:

```
class PrintConsumer(SyncConsumer):
    def test_print(self, message):
        print("Test: " + message["text"])
```

Once you've hooked up the consumers, all you need to do is run a process that will handle them. In lieu of a protocol server - as there are no connections involved here - Channels instead provides you this with the `runworker` command:

```
./manage.py runworker thumbnails-generate thumbnails-delete
```

Note that `runworker` will only listen to the channels you pass it on the command line. If you do not include a channel, or forget to run the worker, your events will not be received and acted upon.

2.12 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 optional extra requirement for a Channels project is to provision a *channel layer*. Both steps are covered below.

2.12.1 Configuring the ASGI application

The one setting that Channels needs to run is `ASGI_APPLICATION`, which tells Channels what the *root application* of your project is. As discussed in *Routing*, this is almost certainly going to be your top-level (Protocol Type) router.

It should be a dotted path to the instance of the router; this is generally going to be in a file like `myproject/routing.py`:

```
ASGI_APPLICATION = "myproject.routing.application"
```

2.12.2 Setting up a channel backend

Note: This step is optional. If you aren't using the channel layer, skip this section.

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": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("redis-server-name", 6379)],
        },
    },
}
```

To use the Redis backend you have to install it:

```
pip install -U channels_redis
```

2.12.3 Run protocol servers

In order to talk to the outside world, your Channels/ASGI application needs to be loaded into a *protocol server*. These can be like WSGI servers and run your application in a HTTP mode, but they can also bridge to any number of other protocols (chat protocols, IoT protocols, even radio networks).

All these servers have their own configuration options, but they all have one thing in common - they will want you to pass them an ASGI application to run. Because Django needs to run setup for things like models when it loads in, you can't just pass in the same variable as you configured in `ASGI_APPLICATION` above; you need a bit more code to get Django ready.

In your project directory, you'll already have a file called `wsgi.py` that does this to present Django as a WSGI application. Make a new file alongside it called `asgi.py` and put this in it:

```
"""
ASGI entrypoint. Configures Django and then runs the application
defined in the ASGI_APPLICATION setting.
"""

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

(continues on next page)

(continued from previous page)

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

If you have any customizations in your `wsgi.py` to do additional things on application start, or different ways of loading settings, you can do those in here as well.

Now you have this file, all you need to do is pass the `application` object inside it to your protocol server as the application it should run:

```
daphne -p 8001 myproject.asgi:application
```

2.12.4 HTTP and WebSocket

While ASGI is a general protocol and we can't cover all possible servers here, it's very likely you will want to deploy a Channels project to work over HTTP and potentially WebSocket, so we'll cover that in some more detail.

The Channels project maintains an official ASGI HTTP/WebSocket server, Daphne, and it's this that we'll talk about configuring. Other HTTP/WebSocket ASGI servers are possible and will work just as well provided they follow the spec, but will have different configuration.

You can choose to either use Daphne for all requests - HTTP and WebSocket - or if you are conservative about stability, keep running standard HTTP requests through a WSGI server and use Daphne only for things WSGI cannot do, like HTTP long-polling and WebSockets. If you do split, you'll need to put something in front of Daphne and your WSGI server to work out what requests to send to each (using HTTP path or domain) - that's not covered here, just know you can do it.

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. Make sure you have an `asgi.py` file as outlined above.

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

```
daphne myproject.asgi:application
```

You should run Daphne 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.

You can also specify the port and IP that Daphne binds to:

```
daphne -b 0.0.0.0 -p 8001 myproject.asgi:application
```

You can see more about Daphne and its options on GitHub.

2.13 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 need at least some code changes to upgrade.

2.13.1 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.

2.13.2 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.

2.13.3 Removed Components

The binding framework has been removed entirely - it was a simplistic implementation, and it being in the core package prevented people from exploring their own solutions. It's likely similar concepts and APIs will appear in a third-party (non-official-Django) package as an option for those who want them.

2.13.4 How to Upgrade

While this is not an exhaustive guide, here are some rough rules on how to proceed with an upgrade.

Given the major changes to the architecture and layout of Channels 2, it is likely that upgrading will be a significant rewrite of your code, depending on what you are doing.

It is **not** a drop-in replacement; we would have done this if we could, but changing to `asyncio` and Python 3 made it almost impossible to keep things backwards-compatible, and we wanted to correct some major design decisions.

Function-based consumers and Routing

Channels 1 allowed you to route by event type (e.g. `websocket.connect`) and pass individual functions with routing that looked like this:

```
channel_routing = [
    route("websocket.connect", connect_blog, path=r'^/liveblog/(?P<slug>[^/]+)/stream/
↪$'),
]
```

And function-based consumers that looked like this:

```
def connect_blog(message, slug):
    ...
```

You'll need to convert these to be class-based consumers, as routing is now done once, at connection time, and so all the event handlers have to be together in a single ASGI application. In addition, URL arguments are no longer passed down into the individual functions - instead, they will be provided in `scope` as the key `url_route`, a dict with an `args` key containing a list of positional regex groups and a `kwargs` key with a dict of the named groups.

Routing is also now the main entry point, so you will need to change routing to have a ProtocolTypeRouter with URLRouters nested inside it. See *Routing* for more.

channel_session and enforce_ordering

Any use of the `channel_session` or `enforce_ordering` decorators can be removed; ordering is now always followed as protocols are handled in the same process, and `channel_session` is not needed as the same application instance now handles all traffic from a single client.

Anywhere you stored information in the `channel_session` can be replaced by storing it on `self` inside a consumer.

HTTP sessions and Django auth

All *authentication* and sessions are now done with middleware. You can remove any decorators that handled them, like `http_session`, `channel_session_user` and so on (in fact, there are no decorators in Channels 2 - it's all middleware).

To get auth now, wrap your URLRouter in an `AuthMiddlewareStack`:

```
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack

application = ProtocolTypeRouter({
    "websocket": AuthMiddlewareStack(
        URLRouter([
            ...
        ])
    ),
})
```

You need to replace accesses to `message.http_session` with `self.scope["session"]`, and `message.user` with `self.scope["user"]`. There is no need to do a handoff like `channel_session_user_from_http` any more - just wrap the auth middleware around and the user will be in the scope for the lifetime of the connection.

Channel Layers

Channel layers are now an optional part of Channels, and the interface they need to provide has changed to be async. Only `channels_redis`, formerly known as `asgi_redis`, has been updated to match so far.

Settings are still similar to before, but there is no longer a `ROUTING` key (the base routing is instead defined with `ASGI_APPLICATION`):

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

All consumers have a `self.channel_layer` and `self.channel_name` object that is populated if you've configured a channel layer. Any messages you send to the `channel_name` will now go to the consumer rather than directly to the client - see the *Channel Layers* documentation for more.

The method names are largely the same, but they're all now awaitables rather than synchronous functions, and `send_group` is now `group_send`.

Group objects

Group objects no longer exist; instead you should use the `group_add`, `group_discard`, and `group_send` methods on the `self.channel_layer` object inside of a consumer directly. As an example:

```
from asgiref.sync import async_to_sync

class ChatConsumer(AsyncWebsocketConsumer):

    async def connect(self):
        await self.channel_layer.group_add("chat", self.channel_name)

    async def disconnect(self):
        await self.channel_layer.group_discard("chat", self.channel_name)
```

Delay server

If you used the delay server before to put things on hold for a few seconds, you can now instead use an `AsyncConsumer` and `asyncio.sleep`:

```
class PingConsumer(AsyncConsumer):

    async def websocket_receive(self, message):
        await asyncio.sleep(1)
        await self.send({
            "type": "websocket.send",
            "text": "pong",
        })
```

Testing

The *testing framework* has been entirely rewritten to be async-based.

While this does make writing tests a lot easier and cleaner, it means you must entirely rewrite any consumer tests completely - there is no backwards-compatible interface with the old testing client as it was synchronous. You can read more about the new testing framework in the *testing documentation*.

Also of note is that the live test case class has been renamed from `ChannelLiveServerTestCase` to `ChannelsLiveServerTestCase` - note the extra `s`.

2.14 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('/ws/');
webSocketBridge.listen();
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

CHAPTER 3

Reference

3.1 ASGI

ASGI, or the Asynchronous Server Gateway Interface, is the specification which Channels and Daphne are built upon, designed to untie Channels apps from a specific application server and provide a common way to write application and middleware code.

It's a spiritual successor to WSGI, designed not only run in an asychronous fashion via `asyncio`, but also supporting multiple protocols.

The full ASGI spec can be found at https://github.com/django/asgiref/blob/master/specs/asgi.rst

3.1.1 Summary

An ASGI application is a callable that takes a scope and returns a coroutine callable, that takes receive and send methods. It's usually written as a class:

```
class Application:

    def __init__(self, scope):
        ...

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

The `scope` dict defines the properties of a connection, like its remote IP (for HTTP) or username (for a chat protocol), and the lifetime of a connection. Applications are *instantiated* once per scope - so, for example, once per HTTP request, or once per open WebSocket connection.

Scopes always have a `type` key, which tells you what kind of connection it is and what other keys to expect in the scope (and what sort of messages to expect).

The `receive` awaitable provides events as dicts as they occur, and the `send` awaitable sends events back to the client in a similar dict format.

A *protocol server* sits between the client and your application code, decoding the raw protocol into the scope and event dicts and encoding anything you send back down onto the protocol.

3.1.2 Composability

ASGI applications, like WSGI ones, are designed to be composable, and this includes Channels' routing and middleware components like `ProtocolTypeRouter` and `SessionMiddeware`. These are just ASGI applications that take other ASGI applications as arguments, so you can pass around just one top-level application for a whole Django project and dispatch down to the right consumer based on what sort of connection you're handling.

3.1.3 Protocol Specifications

The basic ASGI spec only outlines the interface for an ASGI app - it does not specify how network protocols are encoded to and from scopes and event dicts. That's the job of protocol specifications:

- HTTP and WebSocket: https://github.com/django/asgiref/blob/master/specs/www.rst

3.2 Channel Layer Specification

Note: Channel layers are now internal only to Channels, and not used as part of ASGI. This spec defines what Channels and applications written using it expect a channel layer to provide.

3.2.1 Abstract

This document outlines a set of standardized definitions for *channels* and a *channel layer* which provides a mechanism to send and receive messages over them. They allow inter-process communication between different processes to help build applications that have messaging and events between different clients.

3.2.2 Overview

Messages

Messages must be a `dict`. Because these messages are sometimes 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

Channels

Channels are identified by a unicode string name consisting only of ASCII letters, ASCII numerical digits, periods (`.`), dashes (`-`) and underscores (`_`), plus an optional type character (see below).

Channels are a first-in, first out queue with at-most-once delivery semantics. They can have multiple writers and multiple readers; only a single reader should get each written message. Implementations must never deliver a message more than once or to more than one reader, and must drop messages if this is necessary to achieve this restriction.

In order to aid with scaling and network architecture, a distinction is made between channels that have multiple readers, *single-reader channels* that are read from a single unknown location, and *process-specific channels* that are read from a single known process.

Normal channel names contain no type characters, and can be routed however the backend wishes; in particular, they do not have to appear globally consistent, and backends may shard their contents out to different servers so that a querying client only sees some portion of the messages. Calling `receive` on these channels does not guarantee that you will get the messages in order or that you will get anything if the channel is non-empty.

Single-reader channel names contain a question mark (`?`) character in order to indicate to the channel layer that it must make these channels appear globally consistent. The `?` is always preceded by the main channel name and followed by a random portion (e.g. `mything.foo?S4Hr2d`).Channel layers may use the random portion to help pin the channel to a server, but reads from this channel by a single process must always be in-order and return messages if the channel is non-empty. These names must be generated by the `new_channel` call.

Process-specific channel names contain an exclamation mark (`!`) that separates a remote and local part. These channels are received differently; only the name up to and including the `!` character is passed to the `receive()` call, and it will receive any message on any channel with that prefix. This allows a process, such as a HTTP terminator, to listen on a single process-specific channel, and then distribute incoming requests to the appropriate client sockets using the local part (the part after the `!`). The local parts must be generated and managed by the process that consumes them. These channels, like single-reader channels, are guaranteed to give any extant messages in order if received from a single process.

Messages should expire after a set time sitting unread in a channel; the recommendation is one minute, though the best value depends on the channel layer and the way it is deployed, and it is recommended that users are allowed to configure the expiry time.

The maximum message size is 1MB if the message were encoded as JSON; if more data than this needs to be transmitted it must be chunked into smaller messages. All channel layers must support messages up to this size, but channel layer users are encouraged to keep well below it.

Extensions

Extensions are functionality that is not required for basic application code and nearly all protocol server code, and so has been made optional in order to enable lightweight channel layers for applications that don't need the full feature set defined here.

The extensions defined here are:

- `groups`: Allows grouping of channels to allow broadcast; see below for more.
- `flush`: Allows easier testing and development with channel layers.

There is potential to add further extensions; these may be defined by a separate specification, or a new version of this specification.

If application code requires an extension, it should check for it as soon as possible, and hard error if it is not provided. Frameworks should encourage optional use of extensions, while attempting to move any extension-not-found errors to process startup rather than message handling.

Asynchronous Support

All channel layers must provide asynchronous (coroutine) methods for their primary endpoints. End-users will be able to achieve synchronous versions using the `asgiref.sync.async_to_sync` wrapper.

Groups

While the basic channel model is sufficient to handle basic application needs, many more advanced uses of asynchronous messaging require notifying many users at once when an event occurs - imagine a live blog, for example, where every viewer should get a long poll response or WebSocket packet when a new entry is posted.

Thus, there is an *optional* groups extension which allows easier broadcast messaging to groups of channels. End-users are free, of course, to use just channel names and direct sending and build their own persistence/broadcast system instead.

Capacity

To provide backpressure, each channel in a channel layer may have a capacity, defined however the layer wishes (it is recommended that it is configurable by the user using keyword arguments to the channel layer constructor, and furthermore configurable per channel name or name prefix).

When a channel is at or over capacity, trying to send() to that channel may raise ChannelFull, which indicates to the sender the channel is over capacity. How the sender wishes to deal with this will depend on context; for example, a web application trying to send a response body will likely wait until it empties out again, while a HTTP interface server trying to send in a request would drop the request and return a 503 error.

Process-local channels must apply their capacity on the non-local part (that is, up to and including the `!` character), and so capacity is shared among all of the "virtual" channels inside it.

Sending to a group never raises ChannelFull; instead, it must silently drop the message if it is over capacity, as per ASGI's at-most-once delivery policy.

3.2.3 Specification Details

A *channel layer* must provide an object with these attributes (all function arguments are positional):

- `coroutine send(channel, message)`, that takes two arguments: the channel to send on, as a unicode string, and the message to send, as a serializable `dict`.
- `coroutine receive(channels, block=False)`, that takes a list of channel names as unicode strings, and returns with either `(None, None)` or `(channel, message)` if a message is available. If `block` is True, then it will not return a message arrives (or optionally, a built-in timeout, but it is valid to block forever if there are no messages); if `block` is false, it will always return immediately. It is perfectly valid to ignore `block` and always return immediately, or after a delay; `block` means that the call can take as long as it likes before returning a message or nothing, not that it must block until it gets one.
- `coroutine new_channel(pattern)`, that takes a unicode string pattern, and returns a new valid channel name that does not already exist, by adding a unicode string after the `!` or `?` character in `pattern`, and checking for existence of that name in the channel layer. The `pattern` must end with `!` or `?` or this function must error. If the character is `!`, making it a process-specific channel, `new_channel` must be called on the same channel layer that intends to read the channel with `receive`; any other channel layer instance may not receive messages on this channel due to client-routing portions of the appended string.
- `MessageTooLarge`, the exception raised when a send operation fails because the encoded message is over the layer's size limit.

- `ChannelFull`, the exception raised when a send operation fails because the destination channel is over capacity.
- `extensions`, a list of unicode string names indicating which extensions this layer provides, or an empty list if it supports none. The possible extensions can be seen in *Extensions*.

A channel layer implementing the `groups` extension must also provide:

- `coroutine group_add(group, channel)`, that takes a `channel` and adds it to the group given by `group`. Both are unicode strings. If the channel is already in the group, the function should return normally.
- `coroutine group_discard(group, channel)`, that removes the `channel` from the `group` if it is in it, and does nothing otherwise.
- `coroutine group_send(group, message)`, that takes two positional arguments; the group to send to, as a unicode string, and the message to send, as a serializable `dict`. It may raise MessageTooLarge but cannot raise ChannelFull.
- `group_expiry`, an integer number of seconds that specifies how long group membership is valid for after the most recent `group_add` call (see *Persistence* below)

A channel layer implementing the `flush` extension must also provide:

- `coroutine flush()`, that resets the channel layer to a blank state, containing no messages and no groups (if the groups extension is implemented). This call must block until the system is cleared and will consistently look empty to any client, if the channel layer is distributed.

Channel Semantics

Channels **must**:

- Preserve ordering of messages perfectly with only a single reader and writer if the channel is a *single-reader* or *process-specific* channel.
- Never deliver a message more than once.
- Never block on message send (though they may raise ChannelFull or MessageTooLarge)
- Be able to handle messages of at least 1MB in size when encoded as JSON (the implementation may use better encoding or compression, as long as it meets the equivalent size)
- Have a maximum name length of at least 100 bytes.

They should attempt to preserve ordering in all cases as much as possible, but perfect global ordering is obviously not possible in the distributed case.

They are not expected to deliver all messages, but a success rate of at least 99.99% is expected under normal circumstances. Implementations may want to have a "resilience testing" mode where they deliberately drop more messages than usual so developers can test their code's handling of these scenarios.

Persistence

Channel layers do not need to persist data long-term; group memberships only need to live as long as a connection does, and messages only as long as the message expiry time, which is usually a couple of minutes.

If a channel layer implements the `groups` extension, it must persist group membership until at least the time when the member channel has a message expire due to non-consumption, after which it may drop membership at any time. If a channel subsequently has a successful delivery, the channel layer must then not drop group membership until another message expires on that channel.

Channel layers must also drop group membership after a configurable long timeout after the most recent `group_add` call for that membership, the default being 86,400 seconds (one day). The value of this timeout is exposed as the `group_expiry` property on the channel layer.

Approximate Global Ordering

While maintaining true global (across-channels) ordering of messages is entirely unreasonable to expect of many implementations, they should strive to prevent busy channels from overpowering quiet channels.

For example, imagine two channels, `busy`, which spikes to 1000 messages a second, and `quiet`, which gets one message a second. There's a single consumer running `receive(['busy', 'quiet'])` which can handle around 200 messages a second.

In a simplistic for-loop implementation, the channel layer might always check `busy` first; it always has messages available, and so the consumer never even gets to see a message from `quiet`, even if it was sent with the first batch of `busy` messages.

A simple way to solve this is to randomize the order of the channel list when looking for messages inside the channel layer; other, better methods are also available, but whatever is chosen, it should try to avoid a scenario where a message doesn't get received purely because another channel is busy.

Strings and Unicode

In this document, and all sub-specifications, *byte string* refers to `str` on Python 2 and `bytes` on Python 3. If this type still supports Unicode codepoints due to the underlying implementation, then any values should be kept within the 0 - 255 range.

Unicode string refers to `unicode` on Python 2 and `str` on 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).

Channel and group names are always unicode strings, with the additional limitation that they only use the following characters:

- ASCII letters
- The digits `0` through `9`
- Hyphen `-`
- Underscore `_`
- Period `.`
- Question mark `?` (only to delineiate single-reader channel names, and only one per name)
- Exclamation mark `!` (only to delineate process-specific channel names, and only one per name)

3.2.4 Copyright

3.3 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.

3.4 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.

3.4.1 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
- channels_redis issues, for the Redis 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).

3.4.2 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.

3.4.3 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.

3.5 Release Notes

3.5.1 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.

3.5.2 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.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 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.

3.5.7 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.

3.5.8 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.

3.5.9 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.

3.5.10 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.

3.5.11 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.

3.5.12 2.0.0 Release Notes

Channels 2.0 is a major rewrite of Channels, introducing a large amount of changes to the fundamental design and architecture of Channels. Notably:

- Data is no longer transported over a channel layer between protocol server and application; instead, applications run inside their protocol servers (like with WSGI).
- To achieve this, the entire core of channels is now built around Python's `asyncio` framework and runs async-native down until it hits either a Django view or a synchronous consumer.
- Python 2.7 and 3.4 are no longer supported.

More detailed information on the changes and tips on how to port your applications can be found in our *What's new in Channels 2?* documentation.

Backwards Incompatible Changes

Channels 2 is regrettably not backwards-compatible at all with Channels 1 applications due to the large amount of re-architecting done to the code and the switch from synchronous to asynchronous runtimes.

A *migration guide* is available, and a lot of the basic concepts are the same, but the basic class structure and imports have changed.

Our apologies for having to make a breaking change like this, but it was the only way to fix some of the fundamental design issues in Channels 1. Channels 1 will continue to receive security and data-loss fixes for the foreseeable future, but no new features will be added.

3.5.13 2.0.1 Release Notes

Channels 2.0.1 is a patch release of channels, adding a couple of small new features and fixing one bug in URL resolution.

As always, when updating Channels make sure to also update its dependencies (`asgiref` and `daphne`) as these also get their own bugfix updates, and some bugs that may appear to be part of Channels are actually in those packages.

New Features

- There are new async versions of the Websocket generic consumers, `AsyncWebsocketConsumer` and `AsyncJsonWebsocketConsumer`. Read more about them in *Consumers*.
- The old `allowed_hosts_only` decorator has been removed (it was accidentally included in the 2.0 release but didn't work) and replaced with a new `OriginValidator` and `AllowedHostsOriginValidator` set of ASGI middleware. Read more in *Security*.

Bugfixes

- A bug in `URLRouter` which didn't allow you to match beyond the first URL in some situations has been resolved, and a test suite was added for URL resolution to prevent it happening again.

Backwards Incompatible Changes

None.

3.5.14 2.0.2 Release Notes

Channels 2.0.2 is a patch release of Channels, fixing a bug in the database connection handling.

As always, when updating Channels make sure to also update its dependencies (`asgiref` and `daphne`) as these also get their own bugfix updates, and some bugs that may appear to be part of Channels are actually in those packages.

New Features

- There is a new `channels.db.database_sync_to_async` wrapper that is like `sync_to_async` but also closes database connections for you. You can read more about usage in *Database Access*.

Bugfixes

- SyncConsumer and all its descendant classes now close database connections when they exit.

Backwards Incompatible Changes

None.