
Channels Documentation

Release 2.0a1

Andrew Godwin

Sep 08, 2017

Contents

1	Projects	3
2	Topics	5
2.1	In Short	5
2.2	Channels Concepts	7
2.3	Installation	11
2.4	Getting Started with Channels	11
2.5	Deploying	23
2.6	Generic Consumers	28
2.7	Routing	33
2.8	Data Binding	34
2.9	Channels WebSocket wrapper	36
2.10	Channel Layer Types	37
2.11	Delay Server	39
2.12	Testing Consumers	40
2.13	Reference	46
2.14	Frequently Asked Questions	49
2.15	ASGI (Asynchronous Server Gateway Interface) Draft Spec	51
2.16	Community Projects	62
2.17	Contributing	62
2.18	Release Notes	64

Channels is a project to make Django able to handle more than just plain HTTP requests, including WebSockets and HTTP2, as well as the ability to run code after a response has been sent for things like thumbnailing or background calculation.

It's an easy-to-understand extension of the Django view model, and easy to integrate and deploy.

First, read our *Channels Concepts* documentation to get an idea of the data model underlying Channels and how they're used inside Django.

Then, read *Getting Started with Channels* to see how to get up and running with WebSockets with only 30 lines of code.

If you want a quick overview, start with *In Short*.

If you are interested in contributing, please read our *Contributing* docs!

Channels is comprised of five packages:

- [Channels](#), the Django integration layer
- [Daphne](#), the HTTP and WebSocket termination server
- [asgiref](#), the base ASGI library/memory backend
- [asgi_redis](#), the Redis channel backend
- [asgi_rabbitmq](#), the RabbitMQ channel backend
- [asgi_ipc](#), the POSIX IPC channel backend

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

In Short

What is Channels?

Channels extends Django to add *a new layer* that allows two important features:

- WebSocket handling, in a way very *similar to normal views*
- Background tasks, running in the same servers as the rest of Django

It allows other things too, but these are the ones you'll use to start with.

How?

It separates Django into two process types:

- One that handles HTTP and WebSockets
- One that runs views, websocket handlers and background tasks (*consumers*)

They communicate via a protocol called *ASGI*, which is similar to WSGI but runs over a network and allows for more protocol types.

Channels does not introduce `asyncio`, `gevent`, or any other `async` code to your Django code; all of your business logic runs synchronously in a worker process or thread.

I have to change how I run Django?

No, all the new stuff is entirely optional. If you want it, however, you'll change from running Django under a WSGI server, to running:

- An ASGI server, probably [Daphne](#)
- Django worker servers, using `manage.py runworker`

- Something to route ASGI requests over, like Redis.

Even when you're running on Channels, it routes all HTTP requests to the Django view system by default, so it works like before.

What else does Channels give me?

Other features include:

- Easy HTTP long-poll support for thousands of clients at once
- Full session and auth support for WebSockets
- Automatic user login for WebSockets based on site cookies
- Built-in primitives for mass triggering of events (chat, live blogs, etc.)
- Zero-downtime deployment with browsers paused while new workers spin up
- Optional low-level HTTP control on a per-URL basis
- Extendability to other protocols or event sources (e.g. WebRTC, raw UDP, SMS)

Does it scale?

Yes, you can run any number of *protocol servers* (ones that serve HTTP and WebSockets) and *worker servers* (ones that run your Django code) to fit your use case.

The ASGI spec allows a number of different *channel layers* to be plugged in between these two components, with different performance characteristics, and it's designed to allow both easy sharding as well as the ability to run separate clusters with their own protocol and worker servers.

Why doesn't it use my favourite message queue?

Channels is deliberately designed to prefer low latency (goal is a few milliseconds) and high throughput over guaranteed delivery, which doesn't match some message queue designs.

Some features, like *guaranteed ordering of messages*, are opt-in as they incur a performance hit, but make it more message queue like.

Do I need to worry about making all my code async-friendly?

No, all your code runs synchronously without any sockets or event loops to block. You can use async code within a Django view or channel consumer if you like - for example, to fetch lots of URLs in parallel - but it doesn't affect the overall deployed site.

What version of Django does it work with?

You can install Channels as a library for Django ≥ 1.8 . It has a few extra dependencies, but these will all be installed if you use `pip`.

Official project

Channels is not in the Django core as initially planned, but it's an official Django project since September 2016. More information about Channels being adopted as an official project are available on the [Django blog](#).

What do I read next?

Start off by reading about the *concepts underlying Channels*, and then move on to read our example-laden *Getting Started guide*.

Channels Concepts

Django's traditional view of the world revolves around requests and responses; a request comes in, Django is fired up to serve it, generates a response to send, and then Django goes away and waits for the next request.

That was fine when the internet was driven by simple browser interactions, but the modern Web includes things like WebSockets and HTTP2 server push, which allow websites to communicate outside of this traditional cycle.

And, beyond that, there are plenty of non-critical tasks that applications could easily offload until after a response has been sent - like saving things into a cache or thumbnailing newly-uploaded images.

It changes the way Django runs to be "event oriented" - rather than just responding to requests, instead Django responds to a wide array of events sent on *channels*. There's still no persistent state - each event handler, or *consumer* as we call them, is called independently in a way much like a view is called.

Let's look at what *channels* are first.

What is a channel?

The core of the system is, unsurprisingly, a datastructure called a *channel*. What is a channel? It is an *ordered, first-in first-out queue* with *message expiry* and *at-most-once delivery to only one listener at a time*.

You can think of it as analogous to a task queue - messages are put onto the channel by *producers*, and then given to just one of the *consumers* listening to that channel.

By *at-most-once* we say that either one consumer gets the message or nobody does (if the channel implementation crashes, let's say). The alternative is *at-least-once*, where normally one consumer gets the message but when things crash it's sent to more than one, which is not the trade-off we want.

There are a couple of other limitations - messages must be made of serializable types, and stay under a certain size limit - but these are implementation details you won't need to worry about until you get to more advanced usage.

The channels have capacity, so a lot of producers can write lots of messages into a channel with no consumers and then a consumer can come along later and will start getting served those queued messages.

If you've used [channels in Go](#): Go channels are reasonably similar to Django ones. The key difference is that Django channels are network-transparent; the implementations of channels we provide are all accessible across a network to consumers and producers running in different processes or on different machines.

Inside a network, we identify channels uniquely by a name string - you can send to any named channel from any machine connected to the same channel backend. If two different machines both write to the `http.request` channel, they're writing into the same channel.

How do we use channels?

So how is Django using those channels? Inside Django you can write a function to consume a channel:

```
def my_consumer(message):  
    pass
```

And then assign a channel to it in the channel routing:

```
channel_routing = {  
    "some-channel": "myapp.consumers.my_consumer",  
}
```

This means that for every message on the channel, Django will call that consumer function with a message object (message objects have a “content” attribute which is always a dict of data, and a “channel” attribute which is the channel it came from, as well as some others).

Instead of having Django run in the traditional request-response mode, Channels changes Django so that it runs in a worker mode - it listens on all channels that have consumers assigned, and when a message arrives on one, it runs the relevant consumer. So rather than running in just a single process tied to a WSGI server, Django runs in three separate layers:

- Interface servers, which communicate between Django and the outside world. This includes a WSGI adapter as well as a separate WebSocket server - this is explained and covered in *Run interface servers*.
- The channel backend, which is a combination of pluggable Python code and a datastore (e.g. Redis, or a shared memory segment) responsible for transporting messages.
- The workers, that listen on all relevant channels and run consumer code when a message is ready.

This may seem relatively simplistic, but that’s part of the design; rather than try and have a full asynchronous architecture, we’re just introducing a slightly more complex abstraction than that presented by Django views.

A view takes a request and returns a response; a consumer takes a channel message and can write out zero to many other channel messages.

Now, let’s make a channel for requests (called `http.request`), and a channel per client for responses (e.g. `http.response.o4F2h2Fd`), where the response channel is a property (`reply_channel`) of the request message. Suddenly, a view is merely another example of a consumer:

```
# Listens on http.request  
def my_consumer(message):  
    # Decode the request from message format to a Request object  
    django_request = AsgiRequest(message)  
    # Run view  
    django_response = view(django_request)  
    # Encode the response into message format  
    for chunk in AsgiHandler.encode_response(django_response):  
        message.reply_channel.send(chunk)
```

In fact, this is how Channels works. The interface servers transform connections from the outside world (HTTP, WebSockets, etc.) into messages on channels, and then you write workers to handle these messages. Usually you leave normal HTTP up to Django’s built-in consumers that plug it into the view/template system, but you can override it to add functionality if you want.

However, the crucial part is that you can run code (and so send on channels) in response to any event - and that includes ones you create. You can trigger on model saves, on other incoming messages, or from code paths inside views and forms. That approach comes in handy for push-style code - where you use WebSockets or HTTP long-polling to notify clients of changes in real time (messages in a chat, perhaps, or live updates in an admin as another user edits something).

Channel Types

There are actually two major uses for channels in this model. The first, and more obvious one, is the dispatching of work to consumers - a message gets added to a channel, and then any one of the workers can pick it up and run the consumer.

The second kind of channel, however, is used for replies. Notably, these only have one thing listening on them - the interface server. Each reply channel is individually named and has to be routed back to the interface server where its client is terminated.

This is not a massive difference - they both still behave according to the core definition of a *channel* - but presents some problems when we're looking to scale things up. We can happily randomly load-balance normal channels across clusters of channel servers and workers - after all, any worker can process the message - but response channels would have to have their messages sent to the channel server they're listening on.

For this reason, Channels treats these as two different *channel types*, and denotes a *reply channel* by having the channel name contain the character ! - e.g. `http.response!f5G3fE21f`. *Normal channels* do not contain it, but along with the rest of the reply channel name, they must contain only the characters `a-z A-Z 0-9 - _`, and be less than 200 characters long.

It's optional for a backend implementation to understand this - after all, it's only important at scale, where you want to shard the two types differently — but it's present nonetheless. For more on scaling, and how to handle channel types if you're writing a backend or interface server, see [Scaling Up](#).

Groups

Because channels only deliver to a single listener, they can't do broadcast; if you want to send a message to an arbitrary group of clients, you need to keep track of which reply channels of those you wish to send to.

If I had a liveblog where I wanted to push out updates whenever a new post is saved, I could register a handler for the `post_save` signal and keep a set of channels (here, using Redis) to send updates to:

```
redis_conn = redis.Redis("localhost", 6379)

@receiver(post_save, sender=BlogUpdate)
def send_update(sender, instance, **kwargs):
    # Loop through all reply channels and send the update
    for reply_channel in redis_conn.smembers("readers"):
        Channel(reply_channel).send({
            "text": json.dumps({
                "id": instance.id,
                "content": instance.content
            })
        })

# Connected to websocket.connect
def ws_connect(message):
    # Add to reader set
    redis_conn.sadd("readers", message.reply_channel.name)
```

While this will work, there's a small problem - we never remove people from the `readers` set when they disconnect. We could add a consumer that listens to `websocket.disconnect` to do that, but we'd also need to have some kind of expiry in case an interface server is forced to quit or loses power before it can send disconnect signals - your code will never see any disconnect notification but the reply channel is completely invalid and messages you send there will sit there until they expire.

Because the basic design of channels is stateless, the channel server has no concept of "closing" a channel if an interface server goes away - after all, channels are meant to hold messages until a consumer comes along (and some

types of interface server, e.g. an SMS gateway, could theoretically serve any client from any interface server).

We don't particularly care if a disconnected client doesn't get the messages sent to the group - after all, it disconnected - but we do care about cluttering up the channel backend tracking all of these clients that are no longer around (and possibly, eventually getting a collision on the reply channel name and sending someone messages not meant for them, though that would likely take weeks).

Now, we could go back into our example above and add an expiring set and keep track of expiry times and so forth, but what would be the point of a framework if it made you add boilerplate code? Instead, Channels implements this abstraction as a core concept called Groups:

```
@receiver(post_save, sender=BlogUpdate)
def send_update(sender, instance, **kwargs):
    Group("liveblog").send({
        "text": json.dumps({
            "id": instance.id,
            "content": instance.content
        })
    })

# Connected to websocket.connect
def ws_connect(message):
    # Add to reader group
    Group("liveblog").add(message.reply_channel)
    # Accept the connection request
    message.reply_channel.send({"accept": True})

# Connected to websocket.disconnect
def ws_disconnect(message):
    # Remove from reader group on clean disconnect
    Group("liveblog").discard(message.reply_channel)
```

Not only do groups have their own `send()` method (which backends can provide an efficient implementation of), they also automatically manage expiry of the group members - when the channel starts having messages expire on it due to non-consumption, we go in and remove it from all the groups it's in as well. Of course, you should still remove things from the group on disconnect if you can; the expiry code is there to catch cases where the disconnect message doesn't make it for some reason.

Groups are generally only useful for reply channels (ones containing the character `!`), as these are unique-per-client, but can be used for normal channels as well if you wish.

Next Steps

That's the high-level overview of channels and groups, and how you should start thinking about them. Remember, Django provides some channels but you're free to make and consume your own, and all channels are network-transparent.

One thing channels do not do, however, is guarantee delivery. If you need certainty that tasks will complete, use a system designed for this with retries and persistence (e.g. Celery), or alternatively make a management command that checks for completion and re-submits a message to the channel if nothing is completed (rolling your own retry logic, essentially).

We'll cover more about what kind of tasks fit well into Channels in the rest of the documentation, but for now, let's progress to *Getting Started with Channels* and writing some code.

Installation

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

```
pip install -U channels
```

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

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

That's it! Once enabled, `channels` will integrate itself into Django and take control of the `runserver` command. See *Getting Started with Channels* 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.

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
```

Getting Started with Channels

(If you haven't yet, make sure you *install Channels*)

Now, let's get to writing some consumers. If you've not read it already, you should read *Channels Concepts*, as it covers the basic description of what channels and groups are, and lays out some of the important implementation patterns and caveats.

First Consumers

When you first run Django with Channels installed, it will be set up in the default layout - where all HTTP requests (on the `http.request` channel) are routed to the Django view layer - nothing will be different to how things worked in the past with a WSGI-based Django, and your views and static file serving (from `runserver` will work as normal)

As a very basic introduction, let's write a consumer that overrides the built-in handling and handles every HTTP request directly. This isn't something you'd usually do in a project, but it's a good illustration of how channels underlie even core Django - it's less of an addition and more adding a whole new layer under the existing view layer.

Make a new project, a new app, and put this in a `consumers.py` file in the app:

```
from django.http import HttpResponse
from channels.handler import AsgiHandler

def http_consumer(message):
    # Make standard HTTP response - access ASGI path attribute directly
    response = HttpResponse("Hello world! You asked for %s" % message.content['path'])
    # Encode that response into message format (ASGI)
    for chunk in AsgiHandler.encode_response(response):
        message.reply_channel.send(chunk)
```

The most important thing to note here is that, because things we send in messages must be JSON serializable, the request and response messages are in a key-value format. You can read more about that format in the [ASGI specification](#), but you don't need to worry about it too much; just know that there's an `AsgiRequest` class that translates from ASGI into Django request objects, and the `AsgiHandler` class handles translation of `HttpResponse` into ASGI messages, which you see used above. Usually, Django's built-in code will do all this for you when you're using normal views.

Now we need to do one more thing, and that's tell Django that this consumer should be tied to the `http.request` channel rather than the default Django view system. This is done in the settings file - in particular, we need to define our default channel layer and what its routing is set to.

Channel routing is a bit like URL routing, and so it's structured similarly - you point the setting at a dict mapping channels to consumer callables. Here's what that looks like:

```
# In settings.py
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgiref.inmemory.ChannelLayer",
        "ROUTING": "myproject.routing.channel_routing",
    },
}
```

```
# In routing.py
from channels.routing import route
channel_routing = [
    route("http.request", "myapp.consumers.http_consumer"),
]
```

Warning: This example, and most of the examples here, use the “in memory” channel layer. This is the easiest to get started with but provides absolutely no cross-process channel transportation, and so can only be used with `runserver`. You'll want to choose another backend (discussed later) to run things in production.

As you can see, this is a little like Django's `DATABASES` setting; there are named channel layers, with a default one called `default`. Each layer needs a channel layer class, some options (if the channel layer needs them), and a routing scheme, which points to a list containing the routing settings. It's recommended you call this `routing.py` and put it alongside `urls.py` in your project, but you can put it wherever you like, as long as the path is correct.

If you start up `python manage.py runserver` and go to `http://localhost:8000`, you'll see that, rather than a default Django page, you get the Hello World response, so things are working. If you don't see a response, check you [installed Channels correctly](#).

Now, that's not very exciting - raw HTTP responses are something Django has been able to do for a long time. Let's try some WebSockets, and make a basic chat server!

We'll start with a simple server that just echoes every message it gets sent back to the same client - no cross-client communication. It's not terribly useful, but it's a good way to start out writing Channels consumers.

Delete that previous consumer and its routing - we'll want the normal Django view layer to serve HTTP requests from now on, which happens if you don't specify a consumer for `http.request` - and make this WebSocket consumer instead:

```
# In consumers.py

def ws_message(message):
    # ASGI WebSocket packet-received and send-packet message types
    # both have a "text" key for their textual data.
    message.reply_channel.send({
        "text": message.content['text'],
    })
```

Hook it up to the `websocket.receive` channel like this:

```
# In routing.py
from channels.routing import route
from myapp.consumers import ws_message

channel_routing = [
    route("websocket.receive", ws_message),
]
```

Now, let's look at what this is doing. It's tied to the `websocket.receive` channel, which means that it'll get a message whenever a WebSocket packet is sent to us by a client.

When it gets that message, it takes the `reply_channel` attribute from it, which is the unique response channel for that client, and sends the same content back to the client using its `send()` method.

Let's test it! Run `runserver`, open a browser, navigate to a page on the server (you can't use any page's console because of origin restrictions), and put the following into the JavaScript console to open a WebSocket and send some data down it (you might need to change the socket address if you're using a development VM or similar)

```
// Note that the path doesn't matter for routing; any WebSocket
// connection gets bumped over to WebSocket consumers
socket = new WebSocket("ws://" + window.location.host + "/chat/");
socket.onmessage = function(e) {
    alert(e.data);
}
socket.onopen = function() {
    socket.send("hello world");
}
// Call onopen directly if socket is already open
if (socket.readyState == WebSocket.OPEN) socket.onopen();
```

You should see an alert come back immediately saying "hello world" - your message has round-tripped through the server and come back to trigger the alert.

Groups

Now, let's make our echo server into an actual chat server, so people can talk to each other. To do this, we'll use Groups, one of the *core concepts* of Channels, and our fundamental way of doing multi-cast messaging.

To do this, we'll hook up the `websocket.connect` and `websocket.disconnect` channels to add and remove our clients from the `Group` as they connect and disconnect, like this:

```
# In consumers.py
from channels import Group

# Connected to websocket.connect
def ws_add(message):
    # Accept the incoming connection
    message.reply_channel.send({"accept": True})
    # Add them to the chat group
    Group("chat").add(message.reply_channel)

# Connected to websocket.disconnect
def ws_disconnect(message):
    Group("chat").discard(message.reply_channel)
```

Note: You need to explicitly accept WebSocket connections if you override `connect` by sending `accept: True` - you can also reject them at connection time, before they open, by sending `close: True`.

Of course, if you've read through *Channels Concepts*, you'll know that channels added to groups expire out if their messages expire (every channel layer has a message expiry time, usually between 30 seconds and a few minutes, and it's often configurable) - but the `disconnect` handler will get called nearly all of the time anyway.

Note: Channels' design is predicated on expecting and working around failure; it assumes that some small percentage of messages will never get delivered, and so all the core functionality is designed to *expect failure* so that when a message doesn't get delivered, it doesn't ruin the whole system.

We suggest you design your applications the same way - rather than relying on 100% guaranteed delivery, which Channels won't give you, look at each failure case and program something to expect and handle it - be that retry logic, partial content handling, or just having something not work that one time. HTTP requests are just as fallible, and most people's response to that is a generic error page!

Now, that's taken care of adding and removing WebSocket send channels for the `chat` group; all we need to do now is take care of message sending. Instead of echoing the message back to the client like we did above, we'll instead send it to the whole `Group`, which means any client who's been added to it will get the message. Here's all the code:

```
# In consumers.py
from channels import Group

# Connected to websocket.connect
def ws_add(message):
    # Accept the connection
    message.reply_channel.send({"accept": True})
    # Add to the chat group
    Group("chat").add(message.reply_channel)

# Connected to websocket.receive
def ws_message(message):
    Group("chat").send({
        "text": "[user] %s" % message.content['text'],
    })

# Connected to websocket.disconnect
```

```
def ws_disconnect(message):
    Group("chat").discard(message.reply_channel)
```

And what our routing should look like in `routing.py`:

```
from channels.routing import route
from myapp.consumers import ws_add, ws_message, ws_disconnect

channel_routing = [
    route("websocket.connect", ws_add),
    route("websocket.receive", ws_message),
    route("websocket.disconnect", ws_disconnect),
]
```

Note that the `http.request` route is no longer present - if we leave it out, then Django will route HTTP requests to the normal view system by default, which is probably what you want. Even if you have a `http.request` route that matches just a subset of paths or methods, the ones that don't match will still fall through to the default handler, which passes it into URL routing and the views.

With all that code, you now have a working set of a logic for a chat server. Test time! Run `runserver`, open a browser and use that same JavaScript code in the developer console as before

```
// Note that the path doesn't matter right now; any WebSocket
// connection gets bumped over to WebSocket consumers
socket = new WebSocket("ws://" + window.location.host + "/chat/");
socket.onmessage = function(e) {
    alert(e.data);
}
socket.onopen = function() {
    socket.send("hello world");
}
// Call onopen directly if socket is already open
if (socket.readyState == WebSocket.OPEN) socket.onopen();
```

You should see an alert come back immediately saying “hello world” - but this time, you can open another tab and do the same there, and both tabs will receive the message and show an alert. Any incoming message is sent to the `chat` group by the `ws_message` consumer, and both your tabs will have been put into the `chat` group when they connected.

Feel free to put some calls to `print` in your handler functions too, if you like, so you can understand when they're called. You can also use `pdb` and other similar methods you'd use to debug normal Django projects.

Running with Channels

Because Channels takes Django into a multi-process model, you no longer run everything in one process along with a WSGI server (of course, you're still free to do that if you don't want to use Channels). Instead, you run one or more *interface servers*, and one or more *worker servers*, connected by that *channel layer* you configured earlier.

There are multiple kinds of “interface servers”, and each one will service a different type of request - one might do both WebSocket and HTTP requests, while another might act as an SMS message gateway, for example.

These are separate from the “worker servers” where Django will run actual logic, though, and so the *channel layer* transports the content of channels across the network. In a production scenario, you'd usually run *worker servers* as a separate cluster from the *interface servers*, though of course you can run both as separate processes on one machine too.

By default, Django doesn't have a channel layer configured - it doesn't need one to run normal WSGI requests, after all. As soon as you try to add some consumers, though, you'll need to configure one.

In the example above we used the in-memory channel layer implementation as our default channel layer. This just stores all the channel data in a dict in memory, and so isn't actually cross-process; it only works inside `runserver`, as that runs the interface and worker servers in different threads inside the same process. When you deploy to production, you'll need to use a channel layer like the Redis backend `asgi_redis` that works cross-process; see [Channel Layer Types](#) for more.

The second thing, once we have a networked channel backend set up, is to make sure we're running an interface server that's capable of serving WebSockets. To solve this, Channels comes with `daphne`, an interface server that can handle both HTTP and WebSockets at the same time, and then ties this in to run when you run `runserver` - you shouldn't notice any difference from the normal Django `runserver`, though some of the options may be a little different.

(Under the hood, `runserver` is now running `Daphne` in one thread and a worker with `autoreload` in another - it's basically a miniature version of a deployment, but all in one process)

Let's try out the Redis backend - Redis runs on pretty much every machine, and has a very small overhead, which makes it perfect for this kind of thing. Install the `asgi_redis` package using `pip`.

```
pip install asgi_redis
```

and set up your channel layer like this:

```
# In settings.py
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_redis.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("localhost", 6379)],
        },
        "ROUTING": "myproject.routing.channel_routing",
    },
}
```

You'll also need to install the Redis server - there are downloads available for Mac OS and Windows, and it's in pretty much every linux distribution's package manager. For example, on Ubuntu, you can just:

```
sudo apt-get install redis-server
```

Fire up `runserver`, and it'll work as before - unexciting, like good infrastructure should be. You can also try out the cross-process nature; run these two commands in two terminals:

- `manage.py runserver --noworker`
- `manage.py runworker`

As you can probably guess, this disables the worker threads in `runserver` and handles them in a separate process. You can pass `-v 2` to `runworker` if you want to see logging as it runs the consumers.

If Django is in debug mode (`DEBUG=True`), then `runworker` will serve static files, as `runserver` does. Just like a normal Django setup, you'll have to set up your static file serving for when `DEBUG` is turned off.

Persisting Data

Echoing messages is a nice simple example, but it's ignoring the real need for a system like this - persistent state for connections. Let's consider a basic chat site where a user requests a chat room upon initial connection, as part of the URL path (e.g. `wss://host/rooms/room-name`).

The `reply_channel` attribute you've seen before is our unique pointer to the open WebSocket - because it varies between different clients, it's how we can keep track of "who" a message is from. Remember, Channels is network-transparent and can run on multiple workers, so you can't just store things locally in global variables or similar.

Instead, the solution is to persist information keyed by the `reply_channel` in some other data store - sound familiar? This is what Django's session framework does for HTTP requests, using a cookie as the key. Wouldn't it be useful if we could get a session using the `reply_channel` as a key?

Channels provides a `channel_session` decorator for this purpose - it provides you with an attribute called `message.channel_session` that acts just like a normal Django session.

Let's use it now to build a chat server that expects you to pass a chatroom name in the path of your WebSocket request and a query string with your username (we'll ignore auth for now - that's next):

```
# In consumers.py
import json
from channels import import Group
from channels.sessions import channel_session
from urllib.parse import parse_qs

# Connected to websocket.connect
@channel_session
def ws_connect(message, room_name):
    # Accept connection
    message.reply_channel.send({"accept": True})
    # Parse the query string
    params = parse_qs(message.content["query_string"])
    if b"username" in params:
        # Set the username in the session
        message.channel_session["username"] = params[b"username"][0].decode("utf8")
        # Add the user to the room_name group
        Group("chat-%s" % room_name).add(message.reply_channel)
    else:
        # Close the connection.
        message.reply_channel.send({"close": True})

# Connected to websocket.receive
@channel_session
def ws_message(message, room_name):
    Group("chat-%s" % room_name).send({
        "text": json.dumps({
            "text": message["text"],
            "username": message.channel_session["username"],
        }),
    })

# Connected to websocket.disconnect
@channel_session
def ws_disconnect(message, room_name):
    Group("chat-%s" % room_name).discard(message.reply_channel)
```

Update `routing.py` as well:

```
# in routing.py
from channels.routing import route
from myapp.consumers import ws_connect, ws_message, ws_disconnect

channel_routing = [
    route("websocket.connect", ws_connect, path=r"^(?P<room_name>[a-zA-Z0-9_]+)/$",
```

```
route("websocket.receive", ws_message, path=r"^(?P<room_name>[a-zA-Z0-9_]+)/$",
      route("websocket.disconnect", ws_disconnect, path=r"^(?P<room_name>[a-zA-Z0-9_
↪ ]+)/$"),
]
```

If you play around with it from the console (or start building a simple JavaScript chat client that appends received messages to a div), you'll see that you can set a chat room with the initial request.

Authentication

Now, of course, a WebSocket solution is somewhat limited in scope without the ability to live with the rest of your website - in particular, we want to make sure we know what user we're talking to, in case we have things like private chat channels (we don't want a solution where clients just ask for the right channels, as anyone could change the code and just put in private channel names)

It can also save you having to manually make clients ask for what they want to see; if I see you open a WebSocket to my "updates" endpoint, and I know which user you are, I can just auto-add that channel to all the relevant groups (mentions of that user, for example).

Handily, as WebSockets start off using the HTTP protocol, they have a lot of familiar features, including a path, GET parameters, and cookies. We'd like to use these to hook into the familiar Django session and authentication systems; after all, WebSockets are no good unless we can identify who they belong to and do things securely.

In addition, we don't want the interface servers storing data or trying to run authentication; they're meant to be simple, lean, fast processes without much state, and so we'll need to do our authentication inside our consumer functions.

Fortunately, because Channels has an underlying spec for WebSockets and other messages (*ASGI*), it ships with decorators that help you with both authentication and getting the underlying Django session (which is what Django authentication relies on).

Channels can use Django sessions either from cookies (if you're running your websocket server on the same domain as your main site, using something like Daphne), or from a `session_key` GET parameter, which works if you want to keep running your HTTP requests through a WSGI server and offload WebSockets to a second server process on another domain.

You get access to a user's normal Django session using the `http_session` decorator - that gives you a `message.http_session` attribute that behaves just like `request.session`. You can go one further and use `http_session_user` which will provide a `message.user` attribute as well as the session attribute.

Now, one thing to note is that you only get the detailed HTTP information during the `connect` message of a WebSocket connection (you can read more about that in the *ASGI spec*) - this means we're not wasting bandwidth sending the same information over the wire needlessly.

This also means we'll have to grab the user in the connection handler and then store it in the session; thankfully, Channels ships with both a `channel_session_user` decorator that works like the `http_session_user` decorator we mentioned above but loads the user from the `channel` session rather than the `HTTP` session, and a function called `transfer_user` which replicates a user from one session to another. Even better, it combines all of these into a `channel_session_user_from_http` decorator.

Bringing that all together, let's make a chat server where users can only chat to people with the same first letter of their username:

```
# In consumers.py
from channels import Channel, Group
from channels.sessions import channel_session
from channels.auth import channel_session_user, channel_session_user_from_http

# Connected to websocket.connect
```

```

@channel_session_user_from_http
def ws_add(message):
    # Accept connection
    message.reply_channel.send({"accept": True})
    # Add them to the right group
    Group("chat-%s" % message.user.username[0]).add(message.reply_channel)

# Connected to websocket.receive
@channel_session_user
def ws_message(message):
    Group("chat-%s" % message.user.username[0]).send({
        "text": message['text'],
    })

# Connected to websocket.disconnect
@channel_session_user
def ws_disconnect(message):
    Group("chat-%s" % message.user.username[0]).discard(message.reply_channel)

```

If you're just using `runserver` (and so `Daphne`), you can just connect and your cookies should transfer your auth over. If you were running `WebSockets` on a separate domain, you'd have to remember to provide the Django session ID as part of the URL, like this

```
socket = new WebSocket("ws://127.0.0.1:9000/?session_key=abcdefg");
```

You can get the current session key in a template with `{{ request.session.session_key }}`. Note that this can't work with signed cookie sessions - since only HTTP responses can set cookies, it needs a backend it can write to to separately store state.

Security

Unlike AJAX requests, `WebSocket` requests are not limited by the Same-Origin policy. This means you don't have to take any extra steps when you have an HTML page served by host A containing JavaScript code wanting to connect to a `WebSocket` on Host B.

While this can be convenient, it also implies that by default any third-party site can connect to your `WebSocket` application. When you are using the `http_session_user` or the `channel_session_user_from_http` decorator, this connection would be authenticated.

The `WebSocket` specification requires browsers to send the origin of a `WebSocket` request in the HTTP header named `Origin`, but validating that header is left to the server.

You can use the decorator `channels.security.websockets.allowed_hosts_only` on a `websocket.connect` consumer to only allow requests originating from hosts listed in the `ALLOWED_HOSTS` setting:

```

# In consumers.py
from channels import Channel, Group
from channels.sessions import channel_session
from channels.auth import channel_session_user, channel_session_user_from_http
from channels.security.websockets import allowed_hosts_only

# Connected to websocket.connect
@allowed_hosts_only
@channel_session_user_from_http
def ws_add(message):
    # Accept connection
    ...

```

Requests from other hosts or requests with missing or invalid origin header are now rejected.

The name `allowed_hosts_only` is an alias for the class-based decorator `AllowedHostsOnlyOriginValidator`, which inherits from `BaseOriginValidator`. If you have custom requirements for origin validation, create a subclass and overwrite the method `validate_origin(self, message, origin)`. It must return `True` when a message should be accepted, `False` otherwise.

Routing

The `routing.py` file acts very much like Django's `urls.py`, including the ability to route things to different consumers based on `path`, or any other message attribute that's a string (for example, `http.request` messages have a `method` key you could route based on).

Much like `urls`, you route using regular expressions; the main difference is that because the `path` is not special-cased - Channels doesn't know that it's a URL - you have to start patterns with the root `/`, and end includes without a `/` so that when the patterns combine, they work correctly.

Finally, because you're matching against message contents using keyword arguments, you can only use named groups in your regular expressions! Here's an example of routing our chat from above:

```
http_routing = [
    route("http.request", poll_consumer, path=r"^/poll/$", method=r"^POST$"),
]

chat_routing = [
    route("websocket.connect", chat_connect, path=r"^/(?P<room_name>[a-zA-Z0-9_]+)/$"),
    route("websocket.disconnect", chat_disconnect),
]

routing = [
    # You can use a string import path as the first argument as well.
    include(chat_routing, path=r"^/chat"),
    include(http_routing),
]
```

The routing is resolved in order, short-circuiting around the includes if one or more of their matches fails. You don't have to start with the `^` symbol - we use Python's `re.match` function, which starts at the start of a line anyway - but it's considered good practice.

When an include matches part of a message value, it chops off the bit of the value it matched before passing it down to its routes or sub-includes, so you can put the same routing under multiple includes with different prefixes if you like.

Because these matches come through as keyword arguments, we could modify our consumer above to use a room based on URL rather than username:

```
# Connected to websocket.connect
@channel_session_user_from_http
def ws_add(message, room_name):
    # Add them to the right group
    Group("chat-%s" % room_name).add(message.reply_channel)
    # Accept the connection request
    message.reply_channel.send({"accept": True})
```

In the next section, we'll change to sending the `room_name` as a part of the `WebSocket` message - which you might do if you had a multiplexing client - but you could use routing there as well.

Models

So far, we've just been taking incoming messages and rebroadcasting them to other clients connected to the same group, but this isn't that great; really, we want to persist messages to a datastore, and we'd probably like to be able to inject messages into chatrooms from things other than WebSocket client connections (perhaps a built-in bot, or server status messages).

Thankfully, we can just use Django's ORM to handle persistence of messages and easily integrate the send into the save flow of the model, rather than the message receive - that way, any new message saved will be broadcast to all the appropriate clients, no matter where it's saved from.

We'll even take some performance considerations into account: We'll make our own custom channel for new chat messages and move the model save and the chat broadcast into that, meaning the sending process/consumer can move on immediately and not spend time waiting for the database save and the (slow on some backends) `Group.send()` call.

Let's see what that looks like, assuming we have a `ChatMessage` model with `message` and `room` fields:

```
# In consumers.py
from channels import Channel
from channels.sessions import channel_session
from .models import ChatMessage

# Connected to chat-messages
def msg_consumer(message):
    # Save to model
    room = message.content['room']
    ChatMessage.objects.create(
        room=room,
        message=message.content['message'],
    )
    # Broadcast to listening sockets
    Group("chat-%s" % room).send({
        "text": message.content['message'],
    })

# Connected to websocket.connect
@channel_session
def ws_connect(message):
    # Work out room name from path (ignore slashes)
    room = message.content['path'].strip("/")
    # Save room in session and add us to the group
    message.channel_session['room'] = room
    Group("chat-%s" % room).add(message.reply_channel)
    # Accept the connection request
    message.reply_channel.send({"accept": True})

# Connected to websocket.receive
@channel_session
def ws_message(message):
    # Stick the message onto the processing queue
    Channel("chat-messages").send({
        "room": message.channel_session['room'],
        "message": message['text'],
    })

# Connected to websocket.disconnect
@channel_session
def ws_disconnect(message):
```

```
Group("chat-%s" % message.channel_session['room']).discard(message.reply_channel)
```

Update `routing.py` as well:

```
# in routing.py
from channels.routing import route
from myapp.consumers import ws_connect, ws_message, ws_disconnect, msg_consumer

channel_routing = [
    route("websocket.connect", ws_connect),
    route("websocket.receive", ws_message),
    route("websocket.disconnect", ws_disconnect),
    route("chat-messages", msg_consumer),
]
```

Note that we could add messages onto the `chat-messages` channel from anywhere; inside a View, inside another model's `post_save` signal, inside a management command run via `cron`. If we wanted to write a bot, too, we could put its listening logic inside the `chat-messages` consumer, as every message would pass through it.

Enforcing Ordering

There's one final concept we want to introduce you to before you go on to build sites with Channels - consumer ordering.

Because Channels is a distributed system that can have many workers, by default it just processes messages in the order the workers get them off the queue. It's entirely feasible for a WebSocket interface server to send out two `receive` messages close enough together that a second worker will pick up and start processing the second message before the first worker has finished processing the first.

This is particularly annoying if you're storing things in the session in the one consumer and trying to get them in the other consumer - because the `connect` consumer hasn't exited, its session hasn't saved. You'd get the same effect if someone tried to request a view before the login view had finished processing, of course, but HTTP requests usually come in a bit slower from clients.

Channels has a solution - the `enforce_ordering` decorator. All WebSocket messages contain an `order` key, and this decorator uses that to make sure that messages are consumed in the right order. In addition, the `connect` message blocks the socket opening until it's responded to, so you are always guaranteed that `connect` will run before any receives even without the decorator.

The decorator uses `channel_session` to keep track of what numbered messages have been processed, and if a worker tries to run a consumer on an out-of-order message, it raises the `ConsumeLater` exception, which puts the message back on the channel it came from and tells the worker to work on another message.

There's a high cost to using `enforce_ordering`, which is why it's an optional decorator. Here's an example of it being used:

```
# In consumers.py
from channels import Channel, Group
from channels.sessions import channel_session, enforce_ordering
from channels.auth import channel_session_user, channel_session_user_from_http

# Connected to websocket.connect
@channel_session_user_from_http
def ws_add(message):
    # This doesn't need a decorator - it always runs separately
    message.channel_session['sent'] = 0
    # Add them to the right group
```

```

Group("chat").add(message.reply_channel)
# Accept the socket
message.reply_channel.send({"accept": True})

# Connected to websocket.receive
@enforce_ordering
@channel_session_user
def ws_message(message):
    # Without enforce_ordering this wouldn't work right
    message.channel_session['sent'] = message.channel_session['sent'] + 1
    Group("chat").send({
        "text": "%s: %s" % (message.channel_session['sent'], message['text']),
    })

# Connected to websocket.disconnect
@channel_session_user
def ws_disconnect(message):
    Group("chat").discard(message.reply_channel)

```

Generally, the performance (and safety) of your ordering is tied to your session backend's performance. Make sure you choose a session backend wisely if you're going to rely heavily on `enforce_ordering`.

Next Steps

That covers the basics of using Channels; you've seen not only how to use basic channels, but also seen how they integrate with WebSockets, how to use groups to manage logical sets of channels, and how Django's session and authentication systems easily integrate with WebSockets.

We recommend you read through the rest of the reference documentation to see more about what you can do with channels; in particular, you may want to look at our [Deploying](#) documentation to get an idea of how to design and run apps in production environments.

Deploying

Deploying applications using channels requires a few more steps than a normal Django WSGI application, but you have a couple of options as to how to deploy it and how much of your traffic you wish to route through the channel layers.

Firstly, remember that it's an entirely optional part of Django. If you leave a project with the default settings (no `CHANNEL_LAYERS`), it'll just run and work like a normal WSGI app.

When you want to enable channels in production, you need to do three things:

- Set up a channel backend
- Run worker servers
- Run interface servers

You can set things up in one of two ways; either route all traffic through a [HTTP/WebSocket interface server](#), removing the need to run a WSGI server at all; or, just route WebSockets and long-poll HTTP connections to the interface server, and *leave other pages served by a standard WSGI server*.

Routing all traffic through the interface server lets you have WebSockets and long-polling coexist in the same URL tree with no configuration; if you split the traffic up, you'll need to configure a webserver or layer 7 loadbalancer in

front of the two servers to route requests to the correct place based on path or domain. Both methods are covered below.

Setting up a channel backend

The first step is to set up a channel backend. If you followed the *Getting Started with Channels* guide, you will have ended up using the in-memory backend, which is useful for `runserver`, but as it only works inside the same process, useless for actually running separate worker and interface servers.

Instead, take a look at the list of *Channel Layer Types*, and choose one that fits your requirements (additionally, you could use a third-party pluggable backend or write your own - that page also explains the interface and rules a backend has to follow).

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

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

To use the Redis backend you have to install it:

```
pip install -U asgi_redis
```

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

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_ipc.IPCChannelLayer",
        "ROUTING": "my_project.routing.channel_routing",
        "CONFIG": {
            "prefix": "mysite",
        },
    },
}
```

Make sure the same settings file is used across all your workers and interface servers; without it, they won't be able to talk to each other and things will just fail to work.

If you prefer to use RabbitMQ layer, please refer to its [documentation](#). Usually your config will end up like this:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_rabbitmq.RabbitmqChannelLayer",
        "ROUTING": "my_project.routing.channel_routing",
        "CONFIG": {
            "url": "amqp://guest:guest@rabbitmq:5672/%2F",
        },
    },
}
```

Run worker servers

Because the work of running consumers is decoupled from the work of talking to HTTP, WebSocket and other client connections, you need to run a cluster of “worker servers” to do all the processing.

Each server is single-threaded, so it’s recommended you run around one or two per core on each machine; it’s safe to run as many concurrent workers on the same machine as you like, as they don’t open any ports (all they do is talk to the channel backend).

To run a worker server, just run:

```
python manage.py runworker
```

Make sure you run this inside an init system or a program like supervisor that can take care of restarting the process when it exits; the worker server has no retry-on-exit logic, though it will absorb tracebacks from inside consumers and forward them to stderr.

Make sure you keep an eye on how busy your workers are; if they get overloaded, requests will take longer and longer to return as the messages queue up (until the expiry or capacity limit is reached, at which point HTTP connections will start dropping).

In a more complex project, you won’t want all your channels being served by the same workers, especially if you have long-running tasks (if you serve them from the same workers as HTTP requests, there’s a chance long-running tasks could block up all the workers and delay responding to HTTP requests).

To manage this, it’s possible to tell workers to either limit themselves to just certain channel names or ignore specific channels using the `--only-channels` and `--exclude-channels` options. Here’s an example of configuring a worker to only serve HTTP and WebSocket requests:

```
python manage.py runworker --only-channels=http.* --only-channels=websocket.*
```

Or telling a worker to ignore all messages on the “thumbnail” channel:

```
python manage.py runworker --exclude-channels=thumbnail
```

Run interface servers

The final piece of the puzzle is the “interface servers”, the processes that do the work of taking incoming requests and loading them into the channels system.

If you want to support WebSockets, long-poll HTTP requests and other Channels features, you’ll need to run a native ASGI interface server, as the WSGI specification has no support for running these kinds of requests concurrently. We ship with an interface server that we recommend you use called [Daphne](#); it supports WebSockets, long-poll HTTP requests, HTTP/2 and performs quite well.

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

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

To run Daphne, it just needs to be supplied with a channel backend, in much the same way a WSGI server needs to be given an application. First, make sure your project has an `asgi.py` file that looks like this (it should live next to `wsgi.py`):

```
import os
from channels.asgi import get_channel_layer

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "my_project.settings")

channel_layer = get_channel_layer()
```

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

```
daphne my_project.asgi:channel_layer
```

Like `runworker`, you should place this inside an init system or something like `supervisord` to ensure it is re-run if it exits unexpectedly.

If you only run Daphne and no workers, all of your page requests will seem to hang forever; that's because Daphne doesn't have any worker servers to handle the request and it's waiting for one to appear (while `runserver` also uses Daphne, it launches worker threads along with it in the same process). In this scenario, it will eventually time out and give you a 503 error after 2 minutes; you can configure how long it waits with the `--http-timeout` command line argument.

Deploying new versions of code

One of the benefits of decoupling the client connection handling from work processing is that it means you can run new code without dropping client connections; this is especially useful for WebSockets.

Just restart your workers when you have new code (by default, if you send them `SIGTERM` they'll cleanly exit and finish running any in-process consumers), and any queued messages or new connections will go to the new workers. As long as the new code is session-compatible, you can even do staged rollouts to make sure workers on new code aren't experiencing high error rates.

There's no need to restart the WSGI or WebSocket interface servers unless you've upgraded the interface server itself or changed the `CHANNEL_LAYER` setting; none of your code is used by them, and all middleware and code that can customize requests is run on the consumers.

You can even use different Python versions for the interface servers and the workers; the ASGI protocol that channel layers communicate over is designed to be portable across all Python versions.

Running just ASGI

If you are just running Daphne to serve all traffic, then the configuration above is enough where you can just expose it to the Internet and it'll serve whatever kind of request comes in; for a small site, just the one Daphne instance and four or five workers is likely enough.

However, larger sites will need to deploy things at a slightly larger scale, and how you scale things up is different from WSGI; see *Scaling Up*.

Running ASGI alongside WSGI

ASGI and its canonical interface server Daphne are both relatively new, and so you may not wish to run all your traffic through it yet (or you may be using specialized features of your existing WSGI server).

If that's the case, that's fine; you can run Daphne and a WSGI server alongside each other, and only have Daphne serve the requests you need it to (usually WebSocket and long-poll HTTP requests, as these do not fit into the WSGI model).

To do this, just set up your Daphne to serve as we discussed above, and then configure your load-balancer or front HTTP server process to dispatch requests to the correct server - based on either path, domain, or if you can, the Upgrade header.

Dispatching based on path or domain means you'll need to design your WebSocket URLs carefully so you can always tell how to route them at the load-balancer level; the ideal thing is to be able to look for the `Upgrade: WebSocket` header and distinguish connections by this, but not all software supports this and it doesn't help route long-poll HTTP connections at all.

You could also invert this model, and have all connections go to Daphne by default and selectively route some back to the WSGI server, if you have particular URLs or domains you want to use that server on.

Running on a PaaS

To run Django with channels enabled on a Platform-as-a-Service (PaaS), you will need to ensure that your PaaS allows you to run multiple processes at different scaling levels; one group will be running Daphne, as a pure Python application (not a WSGI application), and the other should be running `runworker`.

The PaaS will also either have to provide either its own Redis service or a third process type that lets you run Redis yourself to use the cross-network channel backend; both interface and worker processes need to be able to see Redis, but not each other.

If you are only allowed one running process type, it's possible you could combine both interface server and worker into one process using threading and the in-memory backend; however, this is not recommended for production use as you cannot scale up past a single node without groups failing to work.

Scaling Up

Scaling up a deployment containing channels (and thus running ASGI) is a little different to scaling a WSGI deployment.

The fundamental difference is that the group mechanic requires all servers serving the same site to be able to see each other; if you separate the site up and run it in a few, large clusters, messages to groups will only deliver to WebSockets connected to the same cluster. For some site designs this will be fine, and if you think you can live with this and design around it (which means never designing anything around global notifications or events), this may be a good way to go.

For most projects, you'll need to run a single channel layer at scale in order to achieve proper group delivery. Different backends will scale up differently, but the Redis backend can use multiple Redis servers and spread the load across them using sharding based on consistent hashing.

The key to a channel layer knowing how to scale a channel's delivery is if it contains the `!` character or not, which signifies a single-reader channel. Single-reader channels are only ever connected to by a single process, and so in the Redis case are stored on a single, predictable shard. Other channels are assumed to have many workers trying to read them, and so messages for these can be evenly divided across all shards.

Django channels are still relatively new, and so it's likely that we don't yet know the full story about how to scale things up; we run large load tests to try and refine and improve large-project scaling, but it's no substitute for actual traffic. If you're running channels at scale, you're encouraged to send feedback to the Django team and work with us to hone the design and performance of the channel layer backends, or you're free to make your own; the ASGI specification is comprehensive and comes with a conformance test suite, which should aid in any modification of existing backends or development of new ones.

Generic Consumers

Much like Django's class-based views, Channels has class-based consumers. They provide a way for you to arrange code so it's highly modifiable and inheritable, at the slight cost of it being harder to figure out the execution path.

We recommend you use them if you find them valuable; normal function-based consumers are also entirely valid, however, and may result in more readable code for simpler tasks.

There is one base generic consumer class, `BaseConsumer`, that provides the pattern for method dispatch and is the thing you can build entirely custom consumers on top of, and then protocol-specific subclasses that provide extra utility - for example, the `WebsocketConsumer` provides automatic group management for the connection.

When you use class-based consumers in *routing*, you need to use `route_class` rather than `route`; `route_class` knows how to talk to the class-based consumer and extract the list of channels it needs to listen on from it directly, rather than making you pass it in explicitly.

Here's a routing example:

```
from channels import route, route_class

channel_routing = [
    route_class(consumers.ChatServer, path=r"^/chat/"),
    route("websocket.connect", consumers.ws_connect, path=r"^/$"),
]
```

Class-based consumers are instantiated once for each message they consume, so it's safe to store things on `self` (in fact, `self.message` is the current message by default, and `self.kwargs` are the keyword arguments passed in from the routing).

Base

The `BaseConsumer` class is the foundation of class-based consumers, and what you can inherit from if you wish to build your own entirely from scratch.

You use it like this:

```
from channels.generic import BaseConsumer

class MyConsumer(BaseConsumer):

    method_mapping = {
        "channel.name.here": "method_name",
    }

    def method_name(self, message, **kwargs):
        pass
```

All you need to define is the `method_mapping` dictionary, which maps channel names to method names. The base code will take care of the dispatching for you, and set `self.message` to the current message as well.

If you want to perform more complicated routing, you'll need to override the `dispatch()` and `channel_names()` methods in order to do the right thing; remember, though, your channel names cannot change during runtime and must always be the same for as long as your process runs.

`BaseConsumer` and all other generic consumers that inherit from it provide two instance variables on the class:

- `self.message`, the *Message object* representing the message the consumer was called for.
- `self.kwargs`, keyword arguments from the *Routing*

WebSockets

There are two WebSockets generic consumers; one that provides group management, simpler send/receive methods, and basic method routing, and a subclass which additionally automatically serializes all messages sent and receives using JSON.

The basic WebSocket generic consumer is used like this:

```
from channels.generic.websockets import WebSocketConsumer

class MyConsumer(WebSocketConsumer):

    # Set to True to automatically port users from HTTP cookies
    # (you don't need channel_session_user, this implies it)
    http_user = True

    # Set to True if you want it, else leave it out
    strict_ordering = False

    def connection_groups(self, **kwargs):
        """
        Called to return the list of groups to automatically add/remove
        this connection to/from.
        """
        return ["test"]

    def connect(self, message, **kwargs):
        """
        Perform things on connection start
        """
        # Accept the connection; this is done by default if you don't override
        # the connect function.
        self.message.reply_channel.send({"accept": True})

    def receive(self, text=None, bytes=None, **kwargs):
        """
        Called when a message is received with either text or bytes
        filled out.
        """
        # Simple echo
        self.send(text=text, bytes=bytes)

    def disconnect(self, message, **kwargs):
        """
        Perform things on connection close
        """
        pass
```

You can call `self.send` inside the class to send things to the connection's `reply_channel` automatically. Any group names returned from `connection_groups` are used to add the socket to when it connects and to remove it from when it disconnects; you get keyword arguments too if your URL path, say, affects which group to talk to.

Additionally, the property `self.path` is always set to the current URL path.

The JSON-enabled consumer looks slightly different:

```
from channels.generic.websockets import JsonWebSocketConsumer

class MyConsumer(JsonWebSocketConsumer):
```

```
# Set to True if you want it, else leave it out
strict_ordering = False

def connection_groups(self, **kwargs):
    """
    Called to return the list of groups to automatically add/remove
    this connection to/from.
    """
    return ["test"]

def connect(self, message, **kwargs):
    """
    Perform things on connection start
    """
    pass

def receive(self, content, **kwargs):
    """
    Called when a message is received with decoded JSON content
    """
    # Simple echo
    self.send(content)

def disconnect(self, message, **kwargs):
    """
    Perform things on connection close
    """
    pass

# Optionally provide your own custom json encoder and decoder
# @classmethod
# def decode_json(cls, text):
#     return my_custom_json_decoder(text)
#
# @classmethod
# def encode_json(cls, content):
#     return my_custom_json_encoder(content)
```

For this subclass, `receive` only gets a `content` argument that is the already-decoded JSON as Python datastructures; similarly, `send` now only takes a single argument, which it JSON-encodes before sending down to the client.

Note that this subclass still can't intercept `Group.send()` calls to make them into JSON automatically, but it does provide `self.group_send(name, content)` that will do this for you if you call it explicitly.

`self.close()` is also provided to easily close the `WebSocket` from the server end with an optional status code once you are done with it.

WebSocket Multiplexing

Channels provides a standard way to multiplex different data streams over a single `WebSocket`, called a `Demultiplexer`.

It expects JSON-formatted `WebSocket` frames with two keys, `stream` and `payload`, and will match the `stream` against the mapping to find a channel name. It will then forward the message onto that channel while preserving `reply_channel`, so you can hook consumers up to them directly in the `routing.py` file, and use authentication decorators as you wish.

Example using class-based consumer:

```

from channels.generic.websockets import WebsocketDemultiplexer, JsonWebsocketConsumer

class EchoConsumer(JsonWebsocketConsumer):
    def connect(self, message, multiplexer, **kwargs):
        # Send data with the multiplexer
        multiplexer.send({"status": "I just connected!"})

    def disconnect(self, message, multiplexer, **kwargs):
        print("Stream %s is closed" % multiplexer.stream)

    def receive(self, content, multiplexer, **kwargs):
        # Simple echo
        multiplexer.send({"original_message": content})

class AnotherConsumer(JsonWebsocketConsumer):
    def receive(self, content, multiplexer=None, **kwargs):
        # Some other actions here
        pass

class Demultiplexer(WebsocketDemultiplexer):

    # Wire your JSON consumers here: {stream_name : consumer}
    consumers = {
        "echo": EchoConsumer,
        "other": AnotherConsumer,
    }

    # Optionally provide a custom multiplexer class
    # multiplexer_class = MyCustomJsonEncodingMultiplexer

```

The multiplexer allows the consumer class to be independent of the stream name. It holds the stream name and the demultiplexer on the attributes `stream` and `demultiplexer`.

The *data binding* code will also send out messages to clients in the same format, and you can encode things in this format yourself by using the `WebsocketDemultiplexer.encode` class method.

Sessions and Users

If you wish to use `channel_session` or `channel_session_user` with a class-based consumer, simply set one of the variables in the class body:

```

class MyConsumer(WebsocketConsumer):

    channel_session_user = True

```

This will run the appropriate decorator around your handler methods, and provide `message.channel_session` and `message.user` on the message object - both the one passed in to your handler as an argument as well as `self.message`, as they point to the same instance.

And if you just want to use the user from the django session, add `http_user`:

```

class MyConsumer(WebsocketConsumer):

    http_user = True

```

This will give you `message.user`, which will be the same as `request.user` would be on a regular View.

Applying Decorators

To apply decorators to a class-based consumer, you'll have to wrap a functional part of the consumer; in this case, `get_handler` is likely the place you want to override; like so:

```
class MyConsumer(WebSocketConsumer):  
  
    def get_handler(self, *args, **kwargs):  
        handler = super(MyConsumer, self).get_handler(*args, **kwargs)  
        return your_decorator(handler)
```

You can also use the Django `method_decorator` utility to wrap methods that have `message` as their first positional argument - note that it won't work for more high-level methods, like `WebSocketConsumer.receive`.

As route

Instead of making routes using `route_class` you may use the `as_route` shortcut. This function takes route filters (*Filters*) as kwargs and returns `route_class`. For example:

```
from . import consumers  
  
channel_routing = [  
    consumers.ChatServer.as_route(path=r"^/chat/"),  
]
```

Use the `attrs` dict keyword for dynamic class attributes. For example you have the generic consumer:

```
class MyGenericConsumer(WebSocketConsumer):  
    group = 'default'  
    group_prefix = ''  
  
    def connection_groups(self, **kwargs):  
        return ['_'.join(self.group_prefix, self.group)]
```

You can create consumers with different `group` and `group_prefix` with `attrs`, like so:

```
from . import consumers  
  
channel_routing = [  
    consumers.MyGenericConsumer.as_route(path=r"^/path/1/",  
                                         attrs={'group': 'one', 'group_prefix': 'pre'}  
→),  
    consumers.MyGenericConsumer.as_route(path=r"^/path/2/",  
                                         attrs={'group': 'two', 'group_prefix':  
→ 'public'}),  
]
```

Routing

Routing in Channels is done using a system similar to that in core Django; a list of possible routes is provided, and Channels goes through all routes until a match is found, and then runs the resulting consumer.

The difference comes, however, in the fact that Channels has to route based on more than just URL; channel name is the main thing routed on, and URL path is one of many other optional things you can route on, depending on the protocol (for example, imagine email consumers - they would route on domain or recipient address instead).

The routing Channels takes is just a list of *routing objects* - the three built in ones are `route`, `route_class` and `include`, but any object that implements the routing interface will work:

- A method called `match`, taking a single message as an argument and returning `None` for no match or a tuple of `(consumer, kwargs)` if matched.
- A method called `channel_names`, which returns a set of channel names that will match, which is fed to the channel layer to listen on them.

The three default routing objects are:

- `route`: Takes a channel name, a consumer function, and optional filter keyword arguments.
- `route_class`: Takes a class-based consumer, and optional filter keyword arguments. Channel names are taken from the consumer's `channel_names()` method.
- `include`: Takes either a list or string import path to a routing list, and optional filter keyword arguments.

Filters

Filtering is how you limit matches based on, for example, URLs; you use regular expressions, like so:

```
route("websocket.connect", consumers.ws_connect, path=r"^/chat/$")
```

Note: Unlike Django's URL routing, which strips the first slash of a URL for neatness, Channels includes the first slash, as the routing system is generic and not designed just for URLs.

You can have multiple filters:

```
route("email.receive", comment_response, to_address=r".*@example.com$", subject="^\r\nreply")
```

Multiple filters are always combined with logical AND; that is, you need to match every filter to have the consumer called.

Filters can capture keyword arguments to be passed to your function or your class based consumer methods as a kwarg:

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

You can also specify filters on an `include`:

```
include("blog_includes", path=r"^/liveblog/')
```

When you specify filters on `include`, the matched portion of the attribute is removed for matches inside the `include`; for example, this arrangement matches URLs like `/liveblog/stream/`, because the outside `include` strips off the `/liveblog` part it matches before passing it inside:

```
inner_routes = [
    route("websocket.connect", connect_blog, path=r'^/stream/'),
]

routing = [
    include(inner_routes, path=r'^/liveblog')
]
```

You can also include named capture groups in the filters on an include and they'll be passed to the consumer just like those on `route`; note, though, that if the keyword argument names from the `include` and the `route` clash, the values from `route` will take precedence.

Data Binding

The Channels data binding framework automates the process of tying Django models into frontend views, such as javascript-powered website UIs. It provides a quick and flexible way to generate messages on Groups for model changes and to accept messages that change models themselves.

The main target for the moment is WebSockets, but the framework is flexible enough to be used over any protocol.

What does data binding allow?

Data binding in Channels works two ways:

- Outbound, where model changes made through Django are sent out to listening clients. This includes creation, updates and deletion of instances.
- Inbound, where a standardised message format allows creation, update and deletion of instances to be made by clients sending messages.

Combined, these allow a UI to be designed that automatically updates to reflect new values and reflects across clients. A live blog is easily done using data binding against the post object, for example, or an edit interface can show data live as it's edited by other users.

It has some limitations:

- Signals are used to power outbound binding, so if you change the values of a model outside of Django (or use the `.update()` method on a `QuerySet`), the signals are not triggered and the change will not be sent out. You can trigger changes yourself, but you'll need to source the events from the right place for your system.
- The built-in serializers are based on the built-in Django ones and can only handle certain field types; for more flexibility, you can plug in something like the Django REST Framework serializers.

Getting Started

A single `Binding` subclass will handle outbound and inbound binding for a model, and you can have multiple bindings per model (if you want different formats or permission checks, for example).

You can inherit from the base `Binding` and provide all the methods needed, but we'll focus on the `WebSocket JSON` variant here, as it's the easiest thing to get started and likely close to what you want.

Start off like this:

```

from django.db import models
from channels.binding.websockets import WebSocketBinding

class IntegerValue(models.Model):

    name = models.CharField(max_length=100, unique=True)
    value = models.IntegerField(default=0)

class IntegerValueBinding(WebSocketBinding):

    model = IntegerValue
    stream = "intval"
    fields = ["name", "value"]

    @classmethod
    def group_names(cls, instance):
        return ["intval-updates"]

    def has_permission(self, user, action, pk):
        return True

```

This defines a WebSocket binding - so it knows to send outgoing messages formatted as JSON WebSocket frames - and provides the three things you must always provide:

- `fields` is a whitelist of fields to return in the serialized request. Channels does not default to all fields for security concerns; if you want this, set it to the value `["__all__"]`. As an alternative, `exclude` acts as a blacklist of fields.
- `group_names` returns a list of groups to send outbound updates to based on the instance. For example, you could dispatch posts on different liveblogs to groups that included the parent blog ID in the name; here, we just use a fixed group name. Based on how `group_names` changes as the instance changes, Channels will work out if clients need `create`, `update` or `delete` messages (or if the change is hidden from them).
- `has_permission` returns if an inbound binding update is allowed to actually be carried out on the model. We've been very unsafe and made it always return `True`, but here is where you would check against either Django's or your own permission system to see if the user is allowed that action.

For reference, `action` is always one of the unicode strings `"create"`, `"update"` or `"delete"`. You also supply the *WebSocket Multiplexing* stream name to provide to the client - you must use multiplexing if you use WebSocket data binding.

Just adding the binding like this in a place where it will be imported will get outbound messages sending, but you still need a Consumer that will both accept incoming binding updates and add people to the right Groups when they connect. The WebSocket binding classes use the standard *WebSocket Multiplexing*, so you just need to use that:

```

from channels.generic.websockets import WebSocketDemultiplexer
from .binding import IntegerValueBinding

class Demultiplexer(WebSocketDemultiplexer):

    consumers = {
        "intval": IntegerValueBinding.consumer,
    }

    def connection_groups(self):
        return ["intval-updates"]

```

As well as the standard stream-to-consumer mapping, you also need to set `connection_groups`, a list of groups to put people in when they connect. This should match the logic of `group_names` on your binding - we've used

our fixed group name again. Notice that the binding has a `.consumer` attribute; this is a standard WebSocket-JSON consumer, that the demultiplexer can pass demultiplexed `websocket.receive` messages to.

Tie that into your routing, and you're ready to go:

```
from channels import route_class, route
from .consumers import Demultiplexer
from .models import IntegerValueBinding

channel_routing = [
    route_class(Demultiplexer, path="^/binding/"),
]
```

Frontend Considerations

You can use the standard *Channels WebSocket wrapper* to automatically run demultiplexing, and then tie the events you receive into your frontend framework of choice based on `action`, `pk` and `data`.

Note: Common plugins for data binding against popular JavaScript frameworks are wanted; if you're interested, please get in touch.

Custom Serialization/Protocols

Rather than inheriting from the `WebSocketBinding`, you can inherit directly from the base `Binding` class and implement serialization and deserialization yourself. Until proper reference documentation for this is written, we recommend looking at the source code in `channels/bindings/base.py`; it's reasonably well-commented.

Dealing with Disconnection

Because the data binding Channels ships with has no history of events, it means that when a disconnection happens you may miss events that happen during your offline time. For this reason, it's recommended you reload data directly using an API call once connection has been re-established, don't rely on the live updates for critical functionality, or have UI designs that cope well with missing data (e.g. ones where it's all updates and no creates, so the next update will correct everything).

Channels WebSocket wrapper

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

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

```
{% load staticfiles %}

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

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

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

To process messages

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

To send messages, use the *send* method

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

To demultiplex specific streams

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

To send a message to a specific stream

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

The *WebSocketBridge* instance exposes the underlying *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`

Channel Layer Types

Multiple choices of backend are available, to fill different tradeoffs of complexity, throughput and scalability. You can also write your own backend if you wish; the spec they confirm to is called *ASGI*. Any ASGI-compliant channel layer can be used.

Redis

The Redis layer is the recommended backend to run Channels with, as it supports both high throughput on a single Redis server as well as the ability to run against a set of Redis servers in a sharded mode.

To use the Redis layer, simply install it from PyPI (it lives in a separate package, as we didn't want to force a dependency on the `redis-py` for the main install):

```
pip install -U asgi_redis
```

By default, it will attempt to connect to a Redis server on `localhost:6379`, but you can override this with the `hosts` key in its config:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_redis.RedisChannelLayer",
        "ROUTING": "???",
        "CONFIG": {
            "hosts": [("redis-channel-1", 6379), ("redis-channel-2", 6379)],
        },
    },
}
```

Sharding

The sharding model is based on consistent hashing - in particular, *response channels* are hashed and used to pick a single Redis server that both the interface server and the worker will use.

For normal channels, since any worker can service any channel request, messages are simply distributed randomly among all possible servers, and workers will pick a single server to listen to. Note that if you run more Redis servers than workers, it's very likely that some servers will not have workers listening to them; we recommend you always have at least ten workers for each Redis server to ensure good distribution. Workers will, however, change server periodically (every five seconds or so) so queued messages should eventually get a response.

Note that if you change the set of sharding servers you will need to restart all interface servers and workers with the new set before anything works, and any in-flight messages will be lost (even with persistence, some will); the consistent hashing model relies on all running clients having the same settings. Any misconfigured interface server or worker will drop some or all messages.

RabbitMQ

RabbitMQ layer is comparable to Redis in terms of latency and throughput. It can work with single RabbitMQ node and with Erlang cluster.

You need to install layer package from PyPI:

```
pip install -U asgi_rabbitmq
```

To use it you also need provide link to the virtual host with granted permissions:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_rabbitmq.RabbitmqChannelLayer",
        "ROUTING": "???",
        "CONFIG": {
            "url": "amqp://guest:guest@rabbitmq:5672/%2F",
        },
    },
}
```

This layer has complete [documentation](#) on its own.

IPC

The IPC backend uses POSIX shared memory segments and semaphores in order to allow different processes on the same machine to communicate with each other.

As it uses shared memory, it does not require any additional servers running to get working, and is quicker than any network-based channel layer. However, it can only run between processes on the same machine.

Warning: The IPC layer only communicates between processes on the same machine, and while you might initially be tempted to run a cluster of machines all with their own IPC-based set of processes, this will result in groups not working properly; events sent to a group will only go to those channels that joined the group on the same machine. This backend is for single-machine deployments only.

In-memory

The in-memory layer is only useful when running the protocol server and the worker server in a single process; the most common case of this is `runserver`, where a server thread, this channel layer, and worker thread all co-exist inside the same python process.

Its path is `asgiref.inmemory.ChannelLayer`. If you try and use this channel layer with `runworker`, it will exit, as it does not support cross-process communication.

Writing Custom Channel Layers

The interface channel layers present to Django and other software that communicates over them is codified in a specification called *ASGI*.

Any channel layer that conforms to the *ASGI spec* can be used by Django; just set `BACKEND` to the class to instantiate and `CONFIG` to a dict of keyword arguments to initialize the class with.

Delay Server

Channels has an optional app `channels.delay` that implements the *ASGI Delay Protocol*.

The server is exposed through a custom management command `rundelay` which listens to the `asgi.delay` channel for messages to delay.

Getting Started with Delay

To Install the app add `channels.delay` to `INSTALLED_APPS`:

```
INSTALLED_APPS = (
    ...
    'channels',
    'channels.delay'
)
```

Run `migrate` to create the tables

```
python manage.py migrate
```

Run the delay process to start processing messages

`python manage.py rundelay`

Now you're ready to start delaying messages.

Delaying Messages

To delay a message by a fixed number of milliseconds use the `delay` parameter.

Here's an example:

```
from channels import Channel

delayed_message = {
    'channel': 'example_channel',
    'content': {'x': 1},
    'delay': 10 * 1000
}
# The message will be delayed 10 seconds by the server and then sent
Channel('asgi.delay').send(delayed_message, immediately=True)
```

Testing Consumers

When you want to write unit tests for your new Channels consumers, you'll realize that you can't use the standard Django test client to submit fake HTTP requests - instead, you'll need to submit fake Messages to your consumers, and inspect what Messages they send themselves.

We provide a `TestCase` subclass that sets all of this up for you, however, so you can easily write tests and check what your consumers are sending.

ChannelTestCase

If your tests inherit from the `channels.test.ChannelTestCase` base class, whenever you run tests your channel layer will be swapped out for a captive in-memory layer, meaning you don't need an external server running to run tests.

Moreover, you can inject messages onto this layer and inspect ones sent to it to help test your consumers.

To inject a message onto the layer, simply call `Channel.send()` inside any test method on a `ChannelTestCase` subclass, like so:

```
from channels import Channel
from channels.test import ChannelTestCase

class MyTests(ChannelTestCase):
    def test_a_thing(self):
        # This goes onto an in-memory channel, not the real backend.
        Channel("some-channel-name").send({"foo": "bar"})
```

To receive a message from the layer, you can use `self.get_next_message(channel)`, which handles receiving the message and converting it into a `Message` object for you (if you want, you can call `receive_many` on the underlying channel layer, but you'll get back a raw dict and channel name, which is not what consumers want).

You can use this both to get Messages to send to consumers as their primary argument, as well as to get Messages from channels that consumers are supposed to send on to verify that they did.

You can even pass `require=True` to `get_next_message` to make the test fail if there is no message on the channel (by default, it will return you `None` instead).

Here's an extended example testing a consumer that's supposed to take a value and post the square of it to the "result" channel:

```
from channels import Channel
from channels.test import ChannelTestCase

class MyTests(ChannelTestCase):
    def test_a_thing(self):
        # Inject a message onto the channel to use in a consumer
        Channel("input").send({"value": 33})
        # Run the consumer with the new Message object
        my_consumer(self.get_next_message("input", require=True))
        # Verify there's a result and that it's accurate
        result = self.get_next_message("result", require=True)
        self.assertEqual(result['value'], 1089)
```

Generic Consumers

You can use `ChannelTestCase` to test generic consumers as well. Just pass the message object from `get_next_message` to the constructor of the class. To test replies to a specific channel, use the `reply_channel` property on the `Message` object. For example:

```
from channels import Channel
from channels.test import ChannelTestCase

from myapp.consumers import MyConsumer

class MyTests(ChannelTestCase):

    def test_a_thing(self):
        # Inject a message onto the channel to use in a consumer
        Channel("input").send({"value": 33})
        # Run the consumer with the new Message object
        message = self.get_next_message("input", require=True)
        MyConsumer(message)
        # Verify there's a reply and that it's accurate
        result = self.get_next_message(message.reply_channel.name, require=True)
        self.assertEqual(result['value'], 1089)
```

Groups

You can test `Groups` in the same way as `Channels` inside a `ChannelTestCase`; the entire channel layer is flushed each time a test is run, so it's safe to do group adds and sends during a test. For example:

```
from channels import Group
from channels.test import ChannelTestCase

class MyTests(ChannelTestCase):
    def test_a_thing(self):
        # Add a test channel to a test group
        Group("test-group").add("test-channel")
        # Send to the group
```

```
Group("test-group").send({"value": 42})
# Verify the message got into the destination channel
result = self.get_next_message("test-channel", require=True)
self.assertEqual(result['value'], 42)
```

Clients

For more complicated test suites you can use the `Client` abstraction that provides an easy way to test the full life cycle of messages with a couple of methods: `send` to sending message with given content to the given channel, `consume` to run appointed consumer for the next message, `receive` to getting replies for client. Very often you may need to send and then call a consumer one by one, for this purpose use `send_and_consume` method:

```
from channels.test import ChannelTestCase, Client

class MyTests(ChannelTestCase):

    def test_my_consumer(self):
        client = Client()
        client.send_and_consume('my_internal_channel', {'value': 'my_value'})
        self.assertEqual(client.receive(), {'all is': 'done'})
```

Note: if testing consumers that are expected to close the connection when consuming, set the “`check_accept`” parameter to `False` on “`send_and_consume`”.

You can use `WSClient` for websocket related consumers. It automatically serializes JSON content, manage cookies and headers, give easy access to the session and add ability to authorize your requests. For example:

```
# consumers.py
class RoomConsumer(JsonWebsocketConsumer):
    http_user = True
    groups = ['rooms_watchers']

    def receive(self, content, **kwargs):
        self.send({'rooms': self.message.http_session.get("rooms", [])})
        Channel("rooms_receive").send({'user': self.message.user.id,
                                       'message': content['message']})

# tests.py
from channels import Group
from channels.test import ChannelTestCase, WSClient

class RoomsTests(ChannelTestCase):

    def test_rooms(self):
        client = WSClient()
        user = User.objects.create_user(
            username='test', email='test@test.com', password='123456')
        client.login(username='test', password='123456')

        client.send_and_consume('websocket.connect', path='/rooms/')
        # check that there is nothing to receive
        self.assertIsNone(client.receive())

        # test that the client in the group
```

```

Group(RoomConsumer.groups[0]).send({'text': 'ok'}, immediately=True)
self.assertEqual(client.receive(json=False), 'ok')

client.session['rooms'] = ['test', '1']
client.session.save()

client.send_and_consume('websocket.receive',
                        text={'message': 'hey'},
                        path='/rooms/')

# test 'response'
self.assertEqual(client.receive(), {'rooms': ['test', '1']})

self.assertEqual(self.get_next_message('rooms_receive').content,
                  {'user': user.id, 'message': 'hey'})

# There is nothing to receive
self.assertIsNone(client.receive())

```

Instead of `WSClient.login` method with credentials at arguments you may call `WSClient.force_login` (like at django client) with the user object.

`receive` method by default trying to deserialize json text content of a message, so if you need to pass decoding use `receive(json=False)`, like in the example.

For testing consumers with `enforce_ordering` initialize `HttpClient` with `ordered` flag, but if you wanna use your own order don't use it, use content:

```

client = HttpClient(ordered=True)
client.send_and_consume('websocket.receive', text='1', path='/ws') # order = 0
client.send_and_consume('websocket.receive', text='2', path='/ws') # order = 1
client.send_and_consume('websocket.receive', text='3', path='/ws') # order = 2

# manually
client = HttpClient()
client.send('websocket.receive', content={'order': 0}, text='1')
client.send('websocket.receive', content={'order': 2}, text='2')
client.send('websocket.receive', content={'order': 1}, text='3')

# calling consume 4 time for `waiting` message with order 1
client.consume('websocket.receive')
client.consume('websocket.receive')
client.consume('websocket.receive')
client.consume('websocket.receive')

```

Applying routes

When you need to test your consumers without routes in settings or you want to test your consumers in a more isolate and atomic way, it will be simpler with `apply_routes` contextmanager and decorator for your `ChannelTestCase`. It takes a list of routes that you want to use and overwrites existing routes:

```

from channels.test import ChannelTestCase, WSClient, apply_routes

class MyTests(ChannelTestCase):

    def test_myconsumer(self):
        client = WSClient()

```

```

with apply_routes([MyConsumer.as_route(path='/new')]):
    client.send_and_consume('websocket.connect', '/new')
    self.assertEqual(client.receive(), {'key': 'value'})

```

Test Data binding with WSClient

As you know data binding in channels works in outbound and inbound ways, so that ways tests in different ways and WSClient and apply_routes will help to do this. When you testing outbound consumers you need just import your Binding subclass with specified group_names. At test you can join to one of them, make some changes with target model and check received message. Lets test IntegerValueBinding from *data binding* with creating:

```

from channels.test import ChannelTestCase, WSClient
from channels.signals import consumer_finished

class TestIntegerValueBinding(ChannelTestCase):

    def test_outbound_create(self):
        # We use WSClient because of json encoding messages
        client = WSClient()
        client.join_group("intval-updates") # join outbound binding

        # create target entity
        value = IntegerValue.objects.create(name='fifty', value=50)

        received = client.receive() # receive outbound binding message
        self.assertIsNotNone(received)

        self.assertTrue('payload' in received)
        self.assertTrue('action' in received['payload'])
        self.assertTrue('data' in received['payload'])
        self.assertTrue('name' in received['payload']['data'])
        self.assertTrue('value' in received['payload']['data'])

        self.assertEqual(received['payload']['action'], 'create')
        self.assertEqual(received['payload']['model'], 'values.integervalue')
        self.assertEqual(received['payload']['pk'], value.pk)

        self.assertEqual(received['payload']['data']['name'], 'fifty')
        self.assertEqual(received['payload']['data']['value'], 50)

        # assert that is nothing to receive
        self.assertIsNone(client.receive())

```

There is another situation with inbound binding. It is used with *WebSocket Multiplexing*, So we apply two routes: websocket route for demultiplexer and route with internal consumer for binding itself, connect to websocket endpoint and test different actions. For example:

```

class TestIntegerValueBinding(ChannelTestCase):

    def test_inbound_create(self):
        # check that initial state is empty
        self.assertEqual(IntegerValue.objects.all().count(), 0)

        with apply_routes([Demultiplexer.as_route(path='/'),
                           route("binding.intval", IntegerValueBinding.consumer)]):
            client = WSClient()

```



```

client.send_and_consume('websocket.connect', path='/')
client.send_and_consume('websocket.receive', path='/', text={
    'stream': 'intval',
    'payload': {'action': CREATE, 'data': {'name': 'one', 'value': 1}}
})
# our Demultiplexer route message to the inbound consumer,
# so we need to call this consumer
client.consume('binding.users')

self.assertEqual(IntegerValue.objects.all().count(), 1)
value = IntegerValue.objects.all().first()
self.assertEqual(value.name, 'one')
self.assertEqual(value.value, 1)

```

Multiple Channel Layers

If you want to test code that uses multiple channel layers, specify the alias of the layers you want to mock as the `test_channel_aliases` attribute on the `ChannelTestCase` subclass; by default, only the default layer is mocked.

You can pass an `alias` argument to `get_next_message`, `Client` and `Channel` to use a different layer too.

Live Server Test Case

You can use browser automation libraries like Selenium or Splinter to check your application against real layer installation. First of all provide `TEST_CONFIG` setting to prevent overlapping with running dev environment.

```

CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "asgi_redis.RedisChannelLayer",
        "ROUTING": "my_project.routing.channel_routing",
        "CONFIG": {
            "hosts": [("redis-server-name", 6379)],
        },
        "TEST_CONFIG": {
            "hosts": [("localhost", 6379)],
        },
    },
}

```

Now use `ChannelLiveServerTestCase` for your acceptance tests.

```

from channels.test import ChannelLiveServerTestCase
from splinter import Browser

class IntegrationTest(ChannelLiveServerTestCase):

    def test_browse_site_index(self):

        with Browser() as browser:

            browser.visit(self.live_server_url)
            # the rest of your integration test...

```

In the test above Daphne and Channels worker processes were fired up. These processes run your project against the test database and the default channel layer you specify in the settings. If channel layer support `flush` extension, initial cleanup will be done. So do not run this code against your production environment. `ChannelLiveServerTestCase` can not be used with in memory databases. When using the SQLite database engine the Django tests will by default use an in-memory database. To disable this add the `TEST` setting to the database configuration.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
        'TEST': {
            'NAME': 'testdb.sqlite3'
        }
    }
}
```

When channels infrastructure is ready default web browser will be also started. You can open your website in the real browser which can execute JavaScript and operate on WebSockets. `live_server_ws_url` property is also provided if you decide to run messaging directly from Python.

By default live server test case will serve static files. To disable this feature override `serve_static` class attribute.

```
class IntegrationTest(ChannelLiveServerTestCase):

    serve_static = False

    def test_websocket_message(self):
        # JS and CSS are not available in this test.
        ...
```

Reference

Consumers

When you configure channel routing, the object assigned to a channel should be a callable that takes exactly one positional argument, here called `message`, which is a *message object*. A consumer is any callable that fits this definition.

Consumers are not expected to return anything, and if they do, it will be ignored. They may raise `channels.exceptions.ConsumeLater` to re-insert their current message at the back of the channel it was on, but be aware you can only do this so many time (10 by default) until the message is dropped to avoid deadlocking.

Message

Message objects are what consumers get passed as their only argument. They encapsulate the basic *ASGI* message, which is a `dict`, with extra information. They have the following attributes:

- `content`: The actual message content, as a `dict`. See the *ASGI spec* or protocol message definition document for how this is structured.
- `channel`: A *Channel* object, representing the channel this message was received on. Useful if one consumer handles multiple channels.
- `reply_channel`: A *Channel* object, representing the unique reply channel for this message, or `None` if there isn't one.

- `channel_layer`: A *ChannelLayer* object, representing the underlying channel layer this was received on. This can be useful in projects that have more than one layer to identify where to send messages the consumer generates (you can pass it to the constructor of *Channel* or *Group*)

Channel

Channel objects are a simple abstraction around ASGI channels, which by default are unicode strings. The constructor looks like this:

```
channels.Channel(name, alias=DEFAULT_CHANNEL_LAYER, channel_layer=None)
```

Normally, you'll just call `Channel("my.channel.name")` and it'll make the right thing, but if you're in a project with multiple channel layers set up, you can pass in either the layer alias or the layer object and it'll send onto that one instead. They have the following attributes:

- `name`: The unicode string representing the channel name.
- `channel_layer`: A *ChannelLayer* object, representing the underlying channel layer to send messages on.
- `send(content)`: Sends the `dict` provided as *content* over the channel. The content should conform to the relevant ASGI spec or protocol definition.

Group

Groups represent the underlying *ASGI* group concept in an object-oriented way. The constructor looks like this:

```
channels.Group(name, alias=DEFAULT_CHANNEL_LAYER, channel_layer=None)
```

Like *Channel*, you would usually just pass a name, but can pass a layer alias or object if you want to send on a non-default one. They have the following attributes:

- `name`: The unicode string representing the group name.
- `channel_layer`: A *ChannelLayer* object, representing the underlying channel layer to send messages on.
- `send(content)`: Sends the `dict` provided as *content* to all members of the group.
- `add(channel)`: Adds the given channel (as either a *Channel* object or a unicode string name) to the group. If the channel is already in the group, does nothing.
- `discard(channel)`: Removes the given channel (as either a *Channel* object or a unicode string name) from the group, if it's in the group. Does nothing otherwise.

Channel Layer

These are a wrapper around the underlying *ASGI* channel layers that supplies a routing system that maps channels to consumers, as well as aliases to help distinguish different layers in a project with multiple layers.

You shouldn't make these directly; instead, get them by alias (`default` is the default alias):

```
from channels import channel_layers
layer = channel_layers["default"]
```

They have the following attributes:

- `alias`: The alias of this layer.

- `router`: An object which represents the layer’s mapping of channels to consumers. Has the following attributes:
 - `channels`: The set of channels this router can handle, as unicode strings
 - `match(message)`: Takes a *Message* and returns either a (consumer, kwargs) tuple specifying the consumer to run and the keyword argument to pass that were extracted via routing patterns, or None, meaning there’s no route available.

AsgiRequest

This is a subclass of `django.http.HttpRequest` that provides decoding from ASGI requests, and a few extra methods for ASGI-specific info. The constructor is:

```
channels.handler.AsgiRequest(message)
```

`message` must be an *ASGI* `http.request` format message.

Additional attributes are:

- `reply_channel`, a *Channel* object that represents the `http.response.?` reply channel for this request.
- `message`, the raw ASGI message passed in the constructor.

AsgiHandler

This is a class in `channels.handler` that’s designed to handle the workflow of HTTP requests via ASGI messages. You likely don’t need to interact with it directly, but there are two useful ways you can call it:

- `AsgiHandler(message)` will process the message through the Django view layer and yield one or more response messages to send back to the client, encoded from the Django `HttpResponse`.
- `encode_response(response)` is a classmethod that can be called with a Django `HttpResponse` and will yield one or more ASGI messages that are the encoded response.

Decorators

Channels provides decorators to assist with persisting data and security.

- **`channel_session`**: Provides a session-like object called “`channel_session`” to consumers as a message attribute that will auto-persist across consumers with the same incoming “`reply_channel`” value.

Use this to persist data across the lifetime of a connection.

- **`http_session`**: Wraps a **HTTP or WebSocket connect consumer (or any consumer of messages** that provides a “`cookies`” or “`get`” attribute) to provide a “`http_session`” attribute that behaves like `request.session`; that is, it’s hung off of a per-user session key that is saved in a cookie or passed as the “`session_key`” GET parameter.

It won’t automatically create and set a session cookie for users who don’t have one - that’s what `Session-Middleware` is for, this is a simpler read-only version for more low-level code.

If a message does not have a session we can inflate, the “`session`” attribute will be None, rather than an empty session you can write to.

Does not allow a new session to be set; that must be done via a view. This is only an accessor for any existing session.

- `channel_and_http_session`: Enables both the `channel_session` and `http_session`.

Stores the http session key in the `channel_session` on `websocket.connect` messages. It will then hydrate the `http_session` from that same key on subsequent messages.

- **`allowed_hosts_only`:** Wraps a `WebSocket connect consumer` and ensures the request originates from an allowed host.

Reads the Origin header and only passes request originating from a host listed in `ALLOWED_HOSTS` to the consumer. Requests from other hosts or with a missing or invalid Origin headers are rejected.

Frequently Asked Questions

Why are you doing this rather than just using Tornado/gevent/asyncio/etc.?

They're kind of solving different problems. Tornado, gevent and other in-process async solutions are a way of making a single Python process act asynchronously - doing other things while a HTTP request is going on, or juggling hundreds of incoming connections without blocking on a single one.

Channels is different - all the code you write for consumers runs synchronously. You can do all the blocking filesystem calls and CPU-bound tasks you like and all you'll do is block the one worker you're running on; the other worker processes will just keep on going and handling other messages.

This is partially because Django is all written in a synchronous manner, and rewriting it to all be asynchronous would be a near-impossible task, but also because we believe that normal developers should not have to write asynchronous-friendly code. It's really easy to shoot yourself in the foot; do a tight loop without yielding in the middle, or access a file that happens to be on a slow NFS share, and you've just blocked the entire process.

Channels still uses asynchronous code, but it confines it to the interface layer - the processes that serve HTTP, WebSocket and other requests. These do indeed use asynchronous frameworks (currently, asyncio and Twisted) to handle managing all the concurrent connections, but they're also fixed pieces of code; as an end developer, you'll likely never have to touch them.

All of your work can be with standard Python libraries and patterns and the only thing you need to look out for is worker contention - if you flood your workers with infinite loops, of course they'll all stop working, but that's better than a single thread of execution stopping the entire site.

Why aren't you using node/go/etc. to proxy to Django?

There are a couple of solutions where you can use a more "async-friendly" language (or Python framework) to bridge things like WebSockets to Django - terminate them in (say) a Node process, and then bridge it to Django using either a reverse proxy model, or Redis signalling, or some other mechanism.

The thing is, Channels actually makes it easier to do this if you wish. The key part of Channels is introducing a standardised way to run event-triggered pieces of code, and a standardised way to route messages via named channels that hits the right balance between flexibility and simplicity.

While our interface servers are written in Python, there's nothing stopping you from writing an interface server in another language, providing it follows the same serialisation standards for HTTP/WebSocket/etc. messages. In fact, we may ship an alternative server implementation ourselves at some point.

Why isn't there guaranteed delivery/a retry mechanism?

Channels' design is such that anything is allowed to fail - a consumer can error and not send replies, the channel layer can restart and drop a few messages, a dogpile can happen and a few incoming clients get rejected.

This is because designing a system that was fully guaranteed, end-to-end, would result in something with incredibly low throughput, and almost no problem needs that level of guarantee. If you want some level of guarantee, you can build on top of what Channels provides and add it in (for example, use a database to mark things that need to be cleaned up and resend messages if they aren't after a while, or make idempotent consumers and over-send messages rather than under-send).

That said, it's a good way to design a system to presume any part of it can fail, and design for detection and recovery of that state, rather than hanging your entire livelihood on a system working perfectly as designed. Channels takes this idea and uses it to provide a high-throughput solution that is mostly reliable, rather than a low-throughput one that is *nearly* completely reliable.

Can I run HTTP requests/service calls/etc. in parallel from Django without blocking?

Not directly - Channels only allows a consumer function to listen to channels at the start, which is what kicks it off; you can't send tasks off on channels to other consumers and then *wait on the result*. You can send them off and keep going, but you cannot ever block waiting on a channel in a consumer, as otherwise you'd hit deadlocks, livelocks, and similar issues.

This is partially a design feature - this falls into the class of "difficult async concepts that it's easy to shoot yourself in the foot with" - but also to keep the underlying channels implementation simple. By not allowing this sort of blocking, we can have specifications for channel layers that allows horizontal scaling and sharding.

What you can do is:

- Dispatch a whole load of tasks to run later in the background and then finish your current task - for example, dispatching an avatar thumbnailing task in the avatar upload view, then returning a "we got it!" HTTP response.
- Pass details along to the other task about how to continue, in particular a channel name linked to another consumer that will finish the job, or IDs or other details of the data (remember, message contents are just a dict you can put stuff into). For example, you might have a generic image fetching task for a variety of models that should fetch an image, store it, and pass the resultant ID and the ID of the object you're attaching it to onto a different channel depending on the model - you'd pass the next channel name and the ID of the target object in the message, and then the consumer could send a new message onto that channel name when it's done.
- Have interface servers that perform requests or slow tasks (remember, interface servers are the specialist code which *is* written to be highly asynchronous) and then send their results onto a channel when finished. Again, you can't wait around inside a consumer and block on the results, but you can provide another consumer on a new channel that will do the second half.

How do I associate data with incoming connections?

Channels provides full integration with Django's session and auth system for its WebSockets support, as well as per-websocket sessions for persisting data, so you can easily persist data on a per-connection or per-user basis.

You can also provide your own solution if you wish, keyed off of `message.reply_channel`, which is the unique channel representing the connection, but remember that whatever you store in must be **network-transparent** - storing things in a global variable won't work outside of development.

How do I talk to Channels from my non-Django application?

If you have an external server or script you want to talk to Channels, you have a few choices:

- If it's a Python program, and you've made an `asgi.py` file for your project (see [Deploying](#)), you can import the channel layer directly as `yourproject.asgi.channel_layer` and call `send()` and `receive_many()` on it directly. See the *ASGI spec* for the API the channel layer presents.
- If you just need to send messages in when events happen, you can make a management command that calls `Channel("namehere").send({...})` so your external program can just call `manage.py send_custom_event` (or similar) to send a message. Remember, you can send onto channels from any code in your project.
- If neither of these work, you'll have to communicate with Django over HTTP, WebSocket, or another protocol that your project talks, as normal.

Are channels Python 2, 3 or 2+3?

Django-channels and all of its dependencies are compatible with Python 2.7, 3.4, and higher. This includes the parts of Twisted that some of the Channels packages (like daphne) use.

Why isn't there support for socket.io/SockJS/long poll fallback?

Emulating WebSocket over HTTP long polling requires considerably more effort than terminating WebSockets; some server-side state of the connection must be kept in a place that's accessible from all nodes, so when the new long poll comes in, messages can be replayed onto it.

For this reason, we think it's out of scope for Channels itself, though Channels and Daphne come with first-class support for long-running HTTP connections without taking up a worker thread (you can consume `http.request` and not send a response until later, add the reply channel to groups, and even listen out for the `http.disconnect` channel that tells you when long polls terminate early).

ASGI (Asynchronous Server Gateway Interface) Draft Spec

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

Abstract

This document proposes a standard interface between network protocol servers (particularly web servers) and Python applications, intended to allow handling of multiple common protocol styles (including HTTP, HTTP2, and WebSocket).

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

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

Rationale

The WSGI specification has worked well since it was introduced, and allowed for great flexibility in Python framework and web server choice. However, its design is irrevocably tied to the HTTP-style request/response cycle, and more

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

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

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

Its primary goal is to provide a way to write HTTP/2 and WebSocket code, alongside normal HTTP handling code, however, and part of this design is ensuring there is an easy path to use both existing WSGI servers and applications, as a large majority of Python web usage relies on WSGI and providing an easy path forwards is critical to adoption. Details on that interoperability are covered in *HTTP & WebSocket ASGI Message Format (Draft Spec)*.

The end result of this process has been a specification that includes generalised inter-process communication between Python processes, with a certain set of guarantees and delivery styles that make it suited to low-latency protocol processing and response. It is not intended to replace things like traditional task queues, but it is intended that it could be used for things like distributed systems communication, or as the backbone of a service-oriented architecture for inter-service communication.

Overview

ASGI consists of three different components:

- A *protocol server*, which terminates sockets and translates them into per-event messages
- An *application*, which lives inside a *protocol server*, is instantiated once per socket, and handles event messages as they happen.
- A *channel layer*, which allows communication between different application instances regardless of which process or machine they live in.

Like WSGI, the server hosts the application inside it, and dispatches incoming requests to it in a standardized format. Unlike WSGI, however, applications are instantiated objects rather than simple callables, and they can communicate between themselves, in either a point-to-point or broadcast manner.

The inter-instance communication is an essential part of the design; Python has several existing patterns for handling socket termination nicely with classes (such as the Twisted protocol model), but they are limited to handling inside a single class instance or, with some extra work, a single Python process.

ASGI frames an interface that is a bit more constrained - in that the event messages must be network-encodeable, and follow a certain basic format - to allow cross-process communication and thus have the same abstraction useable from a single-connection application to a large system terminating hundreds of thousands of sockets that are all communicating with each other.

Specification Details

Events and Messages

ASGI decomposes protocols into a series of events that an application must react to. For HTTP, this is as simple as two events in order - `http.request` and `http.disconnect`. For something like a WebSocket, this could be a series of different received frames.

It also lets these events be sourced from other application instances, via either a direct message sent to the application instance's *application channel*, or if the application subscribes its channel to a Group which gets a message broadcast to it.

Each message is a `dict` with a top-level `type` key that contains a unicode string of the message type. Users are free to invent their own message types and send them between application instances for high-level events - for example, a chat application might send chat messages with a user type of `mychat.message`.

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

Communication between application instances, and to other parts of a system, is achieved via an ASGI *channel layer*. This is an abstraction that encapsulates the idea of first-in-first-out, named channels that transport messages over a network via what is essentially a message or service bus.

It's a specification that allows for a set of different implementations to be swapped in based on the end-user's requirements and infrastructure, and is fully specified in `/asgi/channel_layer`.

Each application instance is assigned an *application channel* by the protocol server when it is created, and the protocol server will listen for messages sent to the channel and forward them to the application as event messages.

Application instances are given their application channel and the channel layer object when they are instantiated, so they can store them and use them later on to send messages to other application instances, Groups, or anything else on the channel layer.

Applications

ASGI applications are defined as a callable:

```
application(type, reply, channel_layer, application_channel)
```

- `type`: The type of protocol connecting, e.g. "http" or "websocket".
- `reply`: A callable that takes messages and sends them down the socket
- `channel_layer`: An ASGI channel layer.
- `application_channel`: Channel name that will send messages to this application instance

Which must return another callable:

```
application_instance(message)
```

The first callable is called whenever a new socket comes in to the protocol server, and creates a new *instance* of the application per socket (the object that this first callable returns). That instance is then itself called each time an event message is ready for the application - either from the connected socket, or from another process on the channel layer - and passed the message.

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

```
class Application:

    def __init__(self, type, reply, channel_layer, application_channel):
        ...

    def __call__(self, message):
        ...
```

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

Both the application instance and the `reply` callable passed to the instance on instantiation take messages - these are defined in protocol specifications, and as mentioned above, are dicts. You can use the `type` argument to work out what sort of messages the application should send and receive, or indeed to work out which type-specific Application instance to dispatch to instead.

Handling Protocols

ASGI messages represent two main things - internal application events (for example, a channel might be used to queue thumbnails of previously uploaded videos), and protocol events to/from connected clients.

As such, there are *sub-specifications* that outline encodings to and from ASGI messages for common protocols like HTTP and WebSocket; in particular, the HTTP one covers WSGI/ASGI interoperability. It is recommended that if a protocol becomes commonplace, it should gain standardized formats in a sub-specification of its own.

The message formats are a key part of the specification; without them, the protocol server and application might be able to talk to each other, but may not understand some of what the other is saying. It's equivalent to the standard keys in the `environ` dict for WSGI.

Formats generally cover only the socket-based communications with a protocol server via the application instance's callable and the `reply` callable, and will come with an accompanying `type` value so you can work out which protocol you are handling.

Capacity

Protocol servers cannot service an unbounded number of sockets at once; they need to be able to cap the number of simultaneous application instances.

The implementation of this is left to the individual server, but if a server considers itself over capacity, it should close the incoming socket or request with an appropriate error code and never even instantiate an application instance.

Message Formats

These describe the standardized message formats for the protocols this specification supports. All messages are `dicts` at the top level, and all keys are required unless explicitly marked as optional. If a key is marked optional, a default value is specified, which is to be assumed if the key is missing. Keys are unicode strings.

The one common key across all protocols is `type`, a way to indicate what type of message is being sent. The `type` should be namespaced as `protocol.message_type`, where the `protocol` is the value of `type` given to a new application instance (e.g. `http`), and `message_type` is defined by the protocol spec. Examples of a `type` value include `http.request` and `websocket.send`.

Message formats can be found in the sub-specifications:

HTTP & WebSocket ASGI Message Format (Draft Spec)

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

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

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

HTTP

The HTTP format covers HTTP/1.0, HTTP/1.1 and HTTP/2, as the changes in HTTP/2 are largely on the transport level. A protocol server should give different requests on the same connection different reply channels, and correctly multiplex the responses back into the same stream as they come in. The HTTP version is available as a string in the request message.

HTTP/2 Server Push responses are included, but must be sent prior to the main response, and applications must check for `http_version = 2` before sending them; if a protocol server or connection incapable of Server Push receives these, it must drop them.

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

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

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

Request

Keys:

- `type`: `http.request`
- `http_version`: Unicode string, one of `1.0`, `1.1` or `2`.
- `method`: Unicode string HTTP method name, uppercased.
- `scheme`: Unicode string URL scheme portion (likely `http` or `https`). Optional (but must not be empty), default is `"http"`.
- `path`: Unicode string HTTP path from URL, with percent escapes decoded and UTF8 byte sequences decoded into characters.
- `query_string`: Byte string URL portion after the `?`, not url-decoded.
- `root_path`: Unicode string that indicates the root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional, defaults to `" "`.
- `headers`: A list of `[name, value]` lists, where `name` is the byte string header name, and `value` is the byte string header value. Order of header values must be preserved from the original HTTP request; order

of header names is not important. Duplicates are possible and must be preserved in the message as received. Header names must be lowercased.

- `body`: Body of the request, as a byte string. Optional, defaults to `" "`. If `more_content` is set, treat as start of body and concatenate on further chunks.
- `more_content`: Boolean value signifying if there is additional content to come (as part of a Request Body Chunk message). If `True`, the consuming application should wait until it gets a chunk with this set to `False`. If `False`, the request is complete and should be processed.
- `client`: List of `[host, port]` where `host` is a unicode string of the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer. Optional, defaults to `None`.
- `server`: List of `[host, port]` where `host` is the listening address for this server as a unicode string, and `port` is the integer listening port. Optional, defaults to `None`.

Request Body Chunk

Keys:

- `type`: `http.request.body`
- `content`: Byte string of HTTP body content, will be concatenated onto previously received `content` values and `body` key in Request. Not required if `closed` is `True`, required otherwise.
- `closed`: `True` if the client closed the connection prematurely and the rest of the body. If you receive this, abandon processing of the HTTP request. Optional, defaults to `False`.
- `more_content`: Boolean value signifying if there is additional content to come (as part of a Request Body Chunk message). If `True`, the consuming application should wait until it gets a chunk with this set to `False`. If `False`, the request is complete and should be processed.

Response

Send after any server pushes, and before any response chunks.

Keys:

- `type`: `http.response`
- `status`: Integer HTTP status code.
- `headers`: A list of `[name, value]` lists, where `name` is the byte string header name, and `value` is the byte string header value. Order must be preserved in the HTTP response. Header names must be lowercased. Optional, defaults to an empty list.
- `content`: Byte string of HTTP body content. Optional, defaults to empty string.
- `more_content`: Boolean value signifying if there is additional content to come (as part of a Response Chunk message). If `False`, response will be taken as complete and closed off, and any further messages on the channel will be ignored. Optional, defaults to `False`.

Response Chunk

Keys:

- `type`: `http.response.body`
- `content`: Byte string of HTTP body content, will be concatenated onto previously received `content` values.

- `more_content`: Boolean value signifying if there is additional content to come (as part of a Response Chunk message). If `False`, response will be taken as complete and closed off, and any further messages on the channel will be ignored. Optional, defaults to `False`.

Server Push

Must be sent before any Response or Response Chunk messages.

When a server receives this message, it must treat the Request message in the `request` field of the Server Push as though it were a new HTTP request being received from the network. A server may, if it chooses, apply all of its internal logic to handling this request (e.g. the server may want to try to satisfy the request from a cache). Regardless, if the server is unable to satisfy the request itself it must create a new `http.request` and ask the application to serve the request itself.

This approach limits the amount of knowledge the application has to have about pushed responses: they essentially appear to the application like a normal HTTP request, with the difference being that the application itself triggered the request.

If the remote peer does not support server push, either because it's not a HTTP/2 peer or because `SETTINGS_ENABLE_PUSH` is set to 0, the server must do nothing in response to this message.

Keys:

- `type`: `http.server_push`
- `request`: A Request message. The `body`, `body_channel`, and `reply_channel` fields **MUST** be absent: bodies are not allowed on server-pushed requests, and applications should not create reply channels.

Disconnect

Sent when a HTTP connection is closed. This is mainly useful for long-polling, where you may have added the response channel to a Group or other set of channels you want to trigger a reply to when data arrives.

Keys:

- `type`: `http.disconnect`
- `path`: Unicode string HTTP path from URL, with percent escapes decoded and UTF8 byte sequences decoded into characters.

WebSocket

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

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

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

Connection

Sent when the client initially opens a connection and completes the WebSocket handshake.

This message must be responded to with either an *Accept* message or a *Close* message before the socket will pass `websocket.receive` messages. The protocol server should ideally send this message during the handshake phase of the WebSocket and not complete the handshake until it gets a reply, returning HTTP status code 403 if the connection is denied. If this is not possible, it must buffer WebSocket frames and not send them onto `websocket.receive` until a reply is received, and if the connection is rejected, return WebSocket close code 4403.

Channel: `websocket.connect`

Keys:

- `type`: `websocket.connect`
- `scheme`: Unicode string URL scheme portion (likely `ws` or `wss`). Optional (but must not be empty), default is `ws`.
- `path`: Unicode HTTP path from URL, already urldecoded.
- `query_string`: Byte string URL portion after the `?`. Optional, default is empty string.
- `root_path`: Byte string that indicates the root path this application is mounted at; same as `SCRIPT_NAME` in WSGI. Optional, defaults to empty string.
- `headers`: List of `[name, value]`, where `name` is the header name as byte string and `value` is the header value as a byte string. Order should be preserved from the original HTTP request; duplicates are possible and must be preserved in the message as received. Header names must be lowercased.
- `client`: List of `[host, port]` where `host` is a unicode string of the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer. Optional, defaults to `None`.
- `server`: List of `[host, port]` where `host` is the listening address for this server as a unicode string, and `port` is the integer listening port. Optional, defaults to `None`.
- `order`: The integer value 0.

Receive

Sent when a data frame is received from the client. If `ChannelFull` is raised, you may retry sending it but if it does not send the socket must be closed with `websocket` error code 1013.

Channel: `websocket.receive`

Keys:

- `reply_channel`: Channel name for sending data
- `path`: Path sent during `connect`, sent to make routing easier for apps.
- `bytes`: Byte string of frame content, if it was bytes mode, or `None`.
- `text`: Unicode string of frame content, if it was text mode, or `None`.
- `order`: Order of this frame in the WebSocket stream, starting at 1 (`connect` is 0).

One of `bytes` or `text` must be non-`None`.

Disconnection

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

If `ChannelFull` is raised, then give up attempting to send the message; consumption is not required.

Channel: `websocket.disconnect`

Keys:

- `reply_channel`: Channel name that was used for sending data. Cannot be used to send at this point; provided as a way to identify the connection only.
- `code`: The WebSocket close code (integer), as per the WebSocket spec.
- `path`: Path sent during `connect`, sent to make routing easier for apps.
- `order`: Order of the disconnection relative to the incoming frames' `order` values in `websocket.receive`.

Send/Close/Accept

Sends a data frame to the client and/or closes the connection from the server end and/or accepts a connection. If `ChannelFull` is raised, wait and try again.

If received while the connection is waiting for acceptance after a `connect` message:

- If `accept` is `True`, accept the connection (and send any data provided).
- If `accept` is `False`, reject the connection and do nothing else. If `bytes` or `text` were also present they must be ignored.
- If `bytes` or `text` is present and contains a non-empty value, accept the connection and send the data.
- If `close` is `True` or a positive integer, reject the connection. If `bytes` or `text` is also set and not empty, it should accept the connection, send the frame, then immediately close the connection. Note that any close code integer sent is ignored, as connections are rejected with HTTP's 403 `Forbidden`, unless data is also sent, in which case a full WebSocket close is done with the provided code.

If received while the connection is established:

- If `bytes` or `text` is present, send the data.
- If `close` is `True` or a positive integer, close the connection after any send.
- `accept` is ignored.

Channel: Defined by server, suggested `websocket.send.RANDOMPART!CLIENTID`

Keys:

- `bytes`: Byte string of frame content, if in bytes mode, or `None`.
- `text`: Unicode string of frame content, if in text mode, or `None`.
- `close`: Boolean indicating if the connection should be closed after data is sent, if any. Alternatively, a positive integer specifying the response code. The response code will be 1000 if you pass `True`. Optional, default `False`.
- `accept`: Boolean saying if the connection should be accepted without sending a frame if it is in the handshake phase.

A maximum of one of `bytes` or `text` may be provided. If both are provided, the protocol server should ignore the message entirely.

WSGI Compatibility

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

The adaptability works in two ways:

- **WSGI Server to ASGI:** A WSGI application can be written that transforms `environ` into a Request message, sends it off on the `http.request` channel, and then waits on a generated response channel for a Response message. This has the disadvantage of tying up an entire WSGI thread to poll one channel, but should not be a massive performance drop if there is no backlog on the request channel, and would work fine for an in-process adapter to run a pure-ASGI web application.
- **ASGI to WSGI application:** A small wrapper process is needed that listens on the `http.request` channel, and decodes incoming Request messages into an `environ` dict that matches the WSGI specs, while passing in a `start_response` that stores the values for sending with the first content chunk. Then, the application iterates over the WSGI app, packaging each returned content chunk into a Response or Response Chunk message (if more than one is yielded).

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

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

The `start_response` callable maps similarly to Response:

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

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

TODOs

- Maybe remove `http_version` and replace with `supports_server_push?`

Delay Protocol ASGI Message Format (Draft Spec)

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

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

Delay

Send a message to this channel to delay a message.

Channel: `asgi.delay`

Keys:

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

content specifications for the specified destination channel.

UDP ASGI Message Format (1.0)

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

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

Receive

Sent when a UDP datagram is received.

Channel: `udp.receive`

Keys:

- `reply_channel`: Channel name for sending data, starts with `udp.send!`
- `data`: Byte string of UDP datagram payload.
- `client`: List of `[host, port]` where `host` is a unicode string of the remote host's IPv4 or IPv6 address, and `port` is the remote port as an integer.
- `server`: List of `[host, port]` where `host` is the listening address for this server as a unicode string, and `port` is the integer listening port. Optional, defaults to `None`.

Send

Sent to send out a UDP datagram to a client.

Channel: `udp.send!`

Keys:

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

Protocol Format Guidelines

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

- If the protocol has low-level negotiation, keepalive or other features, handle these within the protocol server and don't expose them in ASGI messages.
- If the protocol is datagram-based, one datagram should equal one ASGI message (unless size is an issue)

Strings and Unicode

In this document, and all sub-specifications, *byte string* refers to `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`), so the distinction is always preserved even across the network.

Copyright

This document has been placed in the public domain.

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.
- `Debugpanel`, 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.

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.

What can I work on?

We're looking for help with the following areas:

- Documentation and tutorial writing

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

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

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

Issues are categorized by difficulty level:

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

They are also classified by type:

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

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

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

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.

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.

Release Notes

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

Demultiplexers 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 persistence; see [Delay Server](#) 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.

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.

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.

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.

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.

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.

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.

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 compatibility

Backwards Incompatible Changes

None.

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.

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.

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.