Channels Documentation

Release 4.0.0

Andrew Godwin

Jun 14, 2023

Contents

Channels is a project that takes Django and extends its abilities beyond HTTP - to handle WebSockets, chat protocols, IoT protocols, and more. It's built on a Python specification called ASGI.

Channels builds upon the native ASGI support in Django. Whilst Django still handles traditional HTTP, Channels gives you the choice to handle other connections in either a synchronous or asynchronous style.

To get started understanding Channels, read our *Introduction*, which will walk through how things work.

Note: This is documentation for the **4.x series** of Channels. If you are looking for documentation for older versions, you can select `3.x`, `2.x`, or `1.x` from the versions selector in the bottom-left corner.

CHAPTER 1

Projects

Channels is comprised of several packages:

- Channels, the Django integration layer
- Daphne, the HTTP and Websocket termination server
- asgiref, the base ASGI library
- channels_redis, the Redis channel layer backend (optional)

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

CHAPTER 2

Topics

2.1 Introduction

Welcome to Channels!

Channels wraps Django's native asynchronous view support, allowing Django projects to handle not only HTTP, but protocols that require long-running connections too - WebSockets, MQTT, chatbots, amateur radio, and more.

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

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

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

2.1.1 Turtles All The Way Down

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

Note: ASGI is the name for the asynchronous server specification that Channels is built on. Like WSGI, it is designed to let you choose between different servers and frameworks rather than being locked into Channels and our server Daphne. You can learn more at https://asgi.readthedocs.io

---

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

application.

We treat HTTP and the existing Django application as part of a bigger whole. Traditional Django views are still there with Channels and still usable - with Django's native ASGI support but you can also write custom HTTP long-polling handling, or WebSocket receivers, and have that code sit alongside your existing code. URL routing, middleware - they are all just ASGI applications.

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

2.1.2 Scopes and Events

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

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

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

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

An example with HTTP:

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

An example with a chatbot:

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

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

2.1.3 What is a Consumer?

A consumer is the basic unit of Channels code. We call it a *consumer* as it *consumes events*, but you can think of it as its own tiny little application. When a request or new socket comes in, Channels will follow its routing table - we'll

look at that in a bit - find the right consumer for that incoming connection, and start up a copy of it.

This means that, unlike Django views, consumers are long-running. They can also be short-running - after all, HTTP requests can also be served by consumers - but they're built around the idea of living for a little while (they live for the duration of a *scope*, as we described above).

A basic consumer looks like this:

```
class ChatConsumer(WebsocketConsumer):

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

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

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

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

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

```
class LogConsumer(WebsocketConsumer):

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

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

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

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

You can read more about consumers in *Consumers*.

2.1.4 Routing and Multiple Protocols

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

```
application = URLRouter([
    path("chat/admin/", AdminChatConsumer.as_asgi()),
    path("chat/", PublicChatConsumer.as_asgi(),
])
```

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

```
class ChattyBotConsumer(SyncConsumer):

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

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

```
application = ProtocolTypeRouter({

    "websocket": URLRouter([
        path("chat/admin/", AdminChatConsumer.as_asgi()),
        path("chat/", PublicChatConsumer.as_asgi()),
    ]),

    "telegram": ChattyBotConsumer.as_asgi(),
})
```

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

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

2.1.5 Cross-Process Communication

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

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

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

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

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

```
# In a consumer
self.channel_layer.send(
    'event',
    {
        'type': 'message',
        'channel': channel,
        'text': text,
    }
)
```

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

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

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

2.1.6 Django Integration

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

```
from django.core.asgi import get_asgi_application
from django.urls import re_path

# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from channels.security.websocket import AllowedHostsOriginValidator

application = ProtocolTypeRouter({
    "http": django_asgi_app,
    "websocket": AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                re_path(r"^front(end)/$", consumers.AsyncChatConsumer.as_asgi()),
            ])
        )
    ),
})
```

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

2.2 Installation

Channels is available on PyPI - to install it run:

```
python -m pip install -U 'channels[daphne]'
```

This will install Channels together with the Daphne ASGI application server. If you wish to use a different application server you can `pip install channels`, without the optional `daphne` add-on.

Once that's done, you should add `daphne` to the beginning of your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = (
    "daphne",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.sites",
    ...
)
```

This will install the Daphne's ASGI version of the `runserver` management command.

You can also add `"channels"` for Channel's `runworker` command.

Then, adjust your project's `asgi.py` file, e.g. `myproject/asgi.py`, to wrap the Django ASGI application:

```
import os

from channels.routing import ProtocolTypeRouter
from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

application = ProtocolTypeRouter({
    "http": django_asgi_app,
    # Just HTTP for now. (We can add other protocols later.)
})
```

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

```
ASGI_APPLICATION = "myproject.asgi.application"
```

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

Note: Please be wary of any other third-party apps that require an overloaded or replacement `runserver` command. Daphne 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, make sure `daphne` is at the top of your `INSTALLED_APPS` or remove the offending app altogether.

2.2.1 Installing the latest development version

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

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

2.3 Tutorial

Channels allows you to use WebSockets and other non-HTTP protocols in your Django site. For example you might want to use WebSockets to allow a page on your site to immediately receive updates from your Django server without using HTTP long-polling or other expensive techniques.

In this tutorial we will build a simple chat server, where you can join an online room, post messages to the room, and have others in the same room see those messages immediately.

2.3.1 Tutorial Part 1: Basic Setup

Note: If you encounter any issue during your coding session, please see the *Troubleshooting* section.

In this tutorial we will build a simple chat server. It will have two pages:

- An index view that lets you type the name of a chat room to join.
- A room view that lets you see messages posted in a particular chat room.

The room view will use a WebSocket to communicate with the Django server and listen for any messages that are posted.

We assume that you are familiar with basic concepts for building a Django site. If not we recommend you complete the Django tutorial first and then come back to this tutorial.

We assume that you have Django installed already. You can tell Django is installed and which version by running the following command in a shell prompt (indicated by the `$` prefix):

```
$ python3 -m django --version
```

We also assume that you have *Channels and Daphne installed* already. You can check by running the following command:

```
$ python3 -c 'import channels; import daphne; print(channels.__version__, daphne.__
↪version__)'
```

This tutorial is written for Channels 4.0, which supports Python 3.7+ and Django 3.2+. If the Channels version does not match, you can refer to the tutorial for your version of Channels by using the version switcher at the bottom left corner of this page, or update Channels to the newest version.

This tutorial also **uses Docker** to install and run Redis. We use Redis as the backing store for the channel layer, which is an optional component of the Channels library that we use in the tutorial. Install Docker from its official website - there are official runtimes for Mac OS and Windows that make it easy to use, and packages for many Linux distributions where it can run natively.

Note: While you can run the standard Django `runserver` without the need for Docker, the channels features we'll be using in later parts of the tutorial will need Redis to run, and we recommend Docker as the easiest way to do this.

Creating a project

If you don't already have a Django project, you will need to create one.

From the command line, `cd` into a directory where you'd like to store your code, then run the following command:

```
$ django-admin startproject mysite
```

This will create a `mysite` directory in your current directory with the following contents:

```
mysite/
    manage.py
    mysite/
        __init__.py
        asgi.py
        settings.py
        urls.py
        wsgi.py
```

Creating the Chat app

We will put the code for the chat server in its own app.

Make sure you're in the same directory as `manage.py` and type this command:

```
$ python3 manage.py startapp chat
```

That'll create a directory `chat`, which is laid out like this:

```
chat/
    __init__.py
    admin.py
    apps.py
    migrations/
        __init__.py
    models.py
    tests.py
    views.py
```

For the purposes of this tutorial, we will only be working with `chat/views.py` and `chat/__init__.py`. So remove all other files from the `chat` directory.

After removing unnecessary files, the `chat` directory should look like:

```
chat/
    __init__.py
    views.py
```

We need to tell our project that the `chat` app is installed. Edit the `mysite/settings.py` file and add `'chat'` to the **INSTALLED_APPS** setting. It'll look like this:

```
# mysite/settings.py
INSTALLED_APPS = [
    'chat',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Add the index view

We will now create the first view, an index view that lets you type the name of a chat room to join.

Create a `templates` directory in your `chat` directory. Within the `templates` directory you have just created, create another directory called `chat`, and within that create a file called `index.html` to hold the template for the index view.

Your chat directory should now look like:

```
chat/
    __init__.py
    templates/
        chat/
            index.html
    views.py
```

Put the following code in `chat/templates/chat/index.html`:

```
<!-- chat/templates/chat/index.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Chat Rooms</title>
</head>
<body>
    What chat room would you like to enter?<br>
    <input id="room-name-input" type="text" size="100"><br>
    <input id="room-name-submit" type="button" value="Enter">

    <script>
        document.querySelector('#room-name-input').focus();
        document.querySelector('#room-name-input').onkeyup = function(e) {
            if (e.key === 'Enter') {  // enter, return
                document.querySelector('#room-name-submit').click();
            }
        };

        document.querySelector('#room-name-submit').onclick = function(e) {
            var roomName = document.querySelector('#room-name-input').value;
            window.location.pathname = '/chat/' + roomName + '/';
        };
    </script>
</body>
</html>
```

Create the view function for the room view. Put the following code in `chat/views.py`:

```
# chat/views.py
from django.shortcuts import render


def index(request):
    return render(request, "chat/index.html")
```

To call the view, we need to map it to a URL - and for this we need a URLconf.

To create a URLconf in the chat directory, create a file called `urls.py`. Your app directory should now look like:

```
chat/
    __init__.py
    templates/
        chat/
            index.html
    urls.py
    views.py
```

In the `chat/urls.py` file include the following code:

```
# chat/urls.py
from django.urls import path

from . import views


urlpatterns = [
    path("", views.index, name="index"),
]
```

The next step is to point the root URLconf at the **chat.urls** module. In `mysite/urls.py`, add an import for **django.urls.include** and insert an **include()** in the **urlpatterns** list, so you have:

```
# mysite/urls.py
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path("chat/", include("chat.urls")),
    path("admin/", admin.site.urls),
]
```

Let's verify that the index view works. Run the following command:

```
$ python3 manage.py runserver
```

You'll see the following output on the command line:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you␣
↪apply the migrations for app(s): admin, auth, contenttypes, sessions.
```

(continues on next page)

(continued from previous page)

```
Run 'python manage.py migrate' to apply them.
August 19, 2022 - 10:05:13
Django version 4.1, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Note: Ignore the warning about unapplied database migrations. We won't be using a database in this tutorial.

Go to http://127.0.0.1:8000/chat/ in your browser and you should see the text "What chat room would you like to enter?" along with a text input to provide a room name.

Type in "lobby" as the room name and press enter. You should be redirected to the room view at http://127.0.0.1:8000/chat/lobby/ but we haven't written the room view yet, so you'll get a "Page not found" error page.

Go to the terminal where you ran the `runserver` command and press Control-C to stop the server.

Integrate the Channels library

So far we've just created a regular Django app; we haven't used the Channels library at all. Now it's time to integrate Channels.

Let's start by creating a routing configuration for Channels. A Channels *routing configuration* is an ASGI application that is similar to a Django URLconf, in that it tells Channels what code to run when an HTTP request is received by the Channels server.

Start by adjusting the `mysite/asgi.py` file to include the following code:

```
# mysite/asgi.py
import os

from channels.routing import ProtocolTypeRouter
from django.core.asgi import get_asgi_application

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

application = ProtocolTypeRouter(
    {
        "http": get_asgi_application(),
        # Just HTTP for now. (We can add other protocols later.)
    }
)
```

Now add the Daphne library to the list of installed apps, in order to enable an ASGI versions of the `runserver` command.

Edit the `mysite/settings.py` file and add `'daphne'` to the top of the `INSTALLED_APPS` setting. It'll look like this:

```
# mysite/settings.py
INSTALLED_APPS = [
    'daphne',
    'chat',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

(continues on next page)

(continued from previous page)

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

You'll also need to point Daphne at the root routing configuration. Edit the `mysite/settings.py` file again and add the following to the bottom of it:

```
# mysite/settings.py
# Daphne
ASGI_APPLICATION = "mysite.asgi.application"
```

With Daphne now in the installed apps, it will take control of the `runserver` command, replacing the standard Django development server with the ASGI compatible version.

Note: The Daphne development server will conflict with any other third-party apps that require an overloaded or replacement runserver command. In order to solve such issues, make sure `daphne` is at the top of your `INSTALLED_APPS`, or remove the offending app altogether.

Let's ensure that the Channels development server is working correctly. Run the following command:

```
$ python3 manage.py runserver
```

You'll see the following output on the command line:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you␣
↪apply the migrations for app(s): admin, auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
August 19, 2022 - 10:20:28
Django version 4.1, using settings 'mysite.settings'
Starting ASGI/Daphne version 3.0.2 development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Notice the line beginning with `Starting ASGI/Daphne ...`. This indicates that the Daphne development server has taken over from the Django development server.

Go to http://127.0.0.1:8000/chat/ in your browser and you should still see the index page that we created before.

Go to the terminal where you ran the `runserver` command and press Control-C to stop the server.

This tutorial continues in *Tutorial 2*.

2.3.2 Tutorial Part 2: Implement a Chat Server

This tutorial begins where *Tutorial 1* left off. We'll get the room page working so that you can chat with yourself and others in the same room.

Add the room view

We will now create the second view, a room view that lets you see messages posted in a particular chat room.

Create a new file `chat/templates/chat/room.html`. Your app directory should now look like:

```
chat/
    __init__.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Create the view template for the room view in `chat/templates/chat/room.html`:

```
<!-- chat/templates/chat/room.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>Chat Room</title>
</head>
<body>
    <textarea id="chat-log" cols="100" rows="20"></textarea><br>
    <input id="chat-message-input" type="text" size="100"><br>
    <input id="chat-message-submit" type="button" value="Send">
    {{ room_name|json_script:"room-name" }}
    <script>
        const roomName = JSON.parse(document.getElementById('room-name').textContent);

        const chatSocket = new WebSocket(
            'ws://'
            + window.location.host
            + '/ws/chat/'
            + roomName
            + '/'
        );

        chatSocket.onmessage = function(e) {
            const data = JSON.parse(e.data);
            document.querySelector('#chat-log').value += (data.message + '\n');
        };

        chatSocket.onclose = function(e) {
            console.error('Chat socket closed unexpectedly');
        };

        document.querySelector('#chat-message-input').focus();
        document.querySelector('#chat-message-input').onkeyup = function(e) {
            if (e.key === 'Enter') {  // enter, return
                document.querySelector('#chat-message-submit').click();
            }
        };

        document.querySelector('#chat-message-submit').onclick = function(e) {
            const messageInputDom = document.querySelector('#chat-message-input');
```

(continues on next page)

(continued from previous page)

```
            const message = messageInputDom.value;
            chatSocket.send(JSON.stringify({
                'message': message
            }));
            messageInputDom.value = '';
        };
    </script>
</body>
</html>
```

Create the view function for the room view in `chat/views.py`:

```
# chat/views.py
from django.shortcuts import render


def index(request):
    return render(request, "chat/index.html")


def room(request, room_name):
    return render(request, "chat/room.html", {"room_name": room_name})
```

Create the route for the room view in `chat/urls.py`:

```
# chat/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path("", views.index, name="index"),
    path("<str:room_name>/", views.room, name="room"),
]
```

Start the Channels development server:

```
$ python3 manage.py runserver
```

Go to http://127.0.0.1:8000/chat/ in your browser and to see the index page.

Type in "lobby" as the room name and press enter. You should be redirected to the room page at http://127.0.0.1:8000/chat/lobby/ which now displays an empty chat log.

Type the message "hello" and press enter. Nothing happens. In particular the message does not appear in the chat log. Why?

The room view is trying to open a WebSocket to the URL `ws://127.0.0.1:8000/ws/chat/lobby/` but we haven't created a consumer that accepts WebSocket connections yet. If you open your browser's JavaScript console, you should see an error that looks like:

```
WebSocket connection to 'ws://127.0.0.1:8000/ws/chat/lobby/' failed: Unexpected␣
↪response code: 500
```

Write your first consumer

When Django accepts an HTTP request, it consults the root URLconf to lookup a view function, and then calls the view function to handle the request. Similarly, when Channels accepts a WebSocket connection, it consults the root routing configuration to lookup a consumer, and then calls various functions on the consumer to handle events from the connection.

We will write a basic consumer that accepts WebSocket connections on the path `/ws/chat/ROOM_NAME/` that takes any message it receives on the WebSocket and echos it back to the same WebSocket.

Note: It is good practice to use a common path prefix like `/ws/` to distinguish WebSocket connections from ordinary HTTP connections because it will make deploying Channels to a production environment in certain configurations easier.

In particular for large sites it will be possible to configure a production-grade HTTP server like nginx to route requests based on path to either (1) a production-grade WSGI server like Gunicorn+Django for ordinary HTTP requests or (2) a production-grade ASGI server like Daphne+Channels for WebSocket requests.

Note that for smaller sites you can use a simpler deployment strategy where Daphne serves all requests - HTTP and WebSocket - rather than having a separate WSGI server. In this deployment configuration no common path prefix like `/ws/` is necessary.

Create a new file `chat/consumers.py`. Your app directory should now look like:

```
chat/
    __init__.py
    consumers.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Put the following code in `chat/consumers.py`:

```
# chat/consumers.py
import json

from channels.generic.websocket import WebsocketConsumer


class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.accept()

    def disconnect(self, close_code):
        pass

    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]

        self.send(text_data=json.dumps({"message": message}))
```

This is a synchronous WebSocket consumer that accepts all connections, receives messages from its client, and echos those messages back to the same client. For now it does not broadcast messages to other clients in the same room.

Note: Channels also supports writing *asynchronous* consumers for greater performance. However any asynchronous consumer must be careful to avoid directly performing blocking operations, such as accessing a Django model. See the *Consumers* reference for more information about writing asynchronous consumers.

We need to create a routing configuration for the `chat` app that has a route to the consumer. Create a new file `chat/routing.py`. Your app directory should now look like:

```
chat/
    __init__.py
    consumers.py
    routing.py
    templates/
        chat/
            index.html
            room.html
    urls.py
    views.py
```

Put the following code in `chat/routing.py`:

```
# chat/routing.py
from django.urls import re_path

from . import consumers

websocket_urlpatterns = [
    re_path(r"ws/chat/(?P<room_name>\w+)/$", consumers.ChatConsumer.as_asgi()),
]
```

We call the `as_asgi()` classmethod in order to get an ASGI application that will instantiate an instance of our consumer for each user-connection. This is similar to Django's `as_view()`, which plays the same role for per-request Django view instances.

(Note we use `re_path()` due to limitations in *URLRouter*.)

The next step is to point the main ASGI configuration at the **chat.routing** module. In `mysite/asgi.py`, import `AuthMiddlewareStack`, `URLRouter`, and `chat.routing`; and insert a `'websocket'` key in the `ProtocolTypeRouter` list in the following format:

```
# mysite/asgi.py
import os

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application

from chat.routing import websocket_urlpatterns

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

import chat.routing
```

(continues on next page)

(continued from previous page)

```
application = ProtocolTypeRouter(
    {
        "http": django_asgi_app,
        "websocket": AllowedHostsOriginValidator(
            AuthMiddlewareStack(URLRouter(websocket_urlpatterns))
        ),
    }
)
```

This root routing configuration specifies that when a connection is made to the Channels development server, the `ProtocolTypeRouter` will first inspect the type of connection. If it is a WebSocket connection (**ws://** or **wss://**), the connection will be given to the `AuthMiddlewareStack`.

The `AuthMiddlewareStack` will populate the connection's **scope** with a reference to the currently authenticated user, similar to how Django's `AuthenticationMiddleware` populates the **request** object of a view function with the currently authenticated user. (Scopes will be discussed later in this tutorial.) Then the connection will be given to the `URLRouter`.

The `URLRouter` will examine the HTTP path of the connection to route it to a particular consumer, based on the provided `url` patterns.

Let's verify that the consumer for the `/ws/chat/ROOM_NAME/` path works. Run migrations to apply database changes (Django's session framework needs the database) and then start the Channels development server:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
$ python3 manage.py runserver
```

Go to the room page at http://127.0.0.1:8000/chat/lobby/ which now displays an empty chat log.

Type the message "hello" and press enter. You should now see "hello" echoed in the chat log.

However if you open a second browser tab to the same room page at http://127.0.0.1:8000/chat/lobby/ and type in a message, the message will not appear in the first tab. For that to work, we need to have multiple instances of the same `ChatConsumer` be able to talk to each other. Channels provides a **channel layer** abstraction that enables this kind of communication between consumers.

Go to the terminal where you ran the `runserver` command and press Control-C to stop the server.

Enable a channel layer

A channel layer is a kind of communication system. It allows multiple consumer instances to talk with each other, and with other parts of Django.

A channel layer provides the following abstractions:

- A **channel** is a mailbox where messages can be sent to. Each channel has a name. Anyone who has the name of a channel can send a message to the channel.
- A **group** is a group of related channels. A group has a name. Anyone who has the name of a group can add/remove a channel to the group by name and send a message to all channels in the group. It is not possible to enumerate what channels are in a particular group.

Every consumer instance has an automatically generated unique channel name, and so can be communicated with via a channel layer.

In our chat application we want to have multiple instances of `ChatConsumer` in the same room communicate with each other. To do that we will have each ChatConsumer add its channel to a group whose name is based on the room name. That will allow ChatConsumers to transmit messages to all other ChatConsumers in the same room.

We will use a channel layer that uses Redis as its backing store. To start a Redis server on port 6379, run the following command (press Control-C to stop it):

```
$ docker run --rm -p 6379:6379 redis:7
```

We need to install channels_redis so that Channels knows how to interface with Redis. Run the following command:

```
$ python3 -m pip install channels_redis
```

Before we can use a channel layer, we must configure it. Edit the `mysite/settings.py` file and add a `CHANNEL_LAYERS` setting to the bottom. It should look like:

```
# mysite/settings.py
# Channels
ASGI_APPLICATION = "mysite.asgi.application"
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```

Note: It is possible to have multiple channel layers configured. However most projects will just use a single `'default'` channel layer.

Let's make sure that the channel layer can communicate with Redis. Open a Django shell and run the following commands:

```
$ python3 manage.py shell
>>> import channels.layers
>>> channel_layer = channels.layers.get_channel_layer()
>>> from asgiref.sync import async_to_sync
>>> async_to_sync(channel_layer.send)('test_channel', {'type': 'hello'})
```

(continues on next page)

(continued from previous page)

```
>>> async_to_sync(channel_layer.receive)('test_channel')
{'type': 'hello'}
```

Type Control-D to exit the Django shell.

Now that we have a channel layer, let's use it in `ChatConsumer`. Put the following code in `chat/consumers.py`, replacing the old code:

```
# chat/consumers.py
import json

from asgiref.sync import async_to_sync
from channels.generic.websocket import WebsocketConsumer


class ChatConsumer(WebsocketConsumer):
    def connect(self):
        self.room_name = self.scope["url_route"]["kwargs"]["room_name"]
        self.room_group_name = f"chat_{self.room_name}"

        # Join room group
        async_to_sync(self.channel_layer.group_add)(
            self.room_group_name, self.channel_name
        )

        self.accept()

    def disconnect(self, close_code):
        # Leave room group
        async_to_sync(self.channel_layer.group_discard)(
            self.room_group_name, self.channel_name
        )

    # Receive message from WebSocket
    def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]

        # Send message to room group
        async_to_sync(self.channel_layer.group_send)(
            self.room_group_name, {"type": "chat.message", "message": message}
        )

    # Receive message from room group
    def chat_message(self, event):
        message = event["message"]

        # Send message to WebSocket
        self.send(text_data=json.dumps({"message": message}))
```

When a user posts a message, a JavaScript function will transmit the message over WebSocket to a ChatConsumer. The ChatConsumer will receive that message and forward it to the group corresponding to the room name. Every ChatConsumer in the same group (and thus in the same room) will then receive the message from the group and forward it over WebSocket back to JavaScript, where it will be appended to the chat log.

Several parts of the new `ChatConsumer` code deserve further explanation:

- **`self.scope["url_route"]["kwargs"]["room_name"]`**

  - Obtains the `'room_name'` parameter from the URL route in `chat/routing.py` that opened the WebSocket connection to the consumer.
  - Every consumer has a *scope* that contains information about its connection, including in particular any positional or keyword arguments from the URL route and the currently authenticated user if any.

- **`self.room_group_name = f"chat_{self.room_name}"`**
  - Constructs a Channels group name directly from the user-specified room name, without any quoting or escaping.
  - Group names may only contain alphanumerics, hyphens, underscores, or periods. Therefore this example code will fail on room names that have other characters.

- **`async_to_sync(self.channel_layer.group_add)(...)`**
  - Joins a group.
  - The `async_to_sync(...)` wrapper is required because ChatConsumer is a synchronous WebsocketConsumer but it is calling an asynchronous channel layer method. (All channel layer methods are asynchronous.)
  - Group names are restricted to ASCII alphanumerics, hyphens, and periods only and are limited to a maximum length of 100 in the default backend. Since this code constructs a group name directly from the room name, it will fail if the room name contains any characters that aren't valid in a group name or exceeds the length limit.

- **`self.accept()`**
  - Accepts the WebSocket connection.
  - If you do not call `accept()` within the `connect()` method then the connection will be rejected and closed. You might want to reject a connection for example because the requesting user is not authorized to perform the requested action.
  - It is recommended that `accept()` be called as the *last* action in `connect()` if you choose to accept the connection.

- **`async_to_sync(self.channel_layer.group_discard)(...)`**
  - Leaves a group.

- **`async_to_sync(self.channel_layer.group_send)`**
  - Sends an event to a group.
  - An event has a special `'type'` key corresponding to the name of the method that should be invoked on consumers that receive the event. This translation is done by replacing `.` with `_`, thus in this example, `chat.message` calls the `chat_message` method.

Let's verify that the new consumer for the `/ws/chat/ROOM_NAME/` path works. To start the Channels development server, run the following command:

```
$ python3 manage.py runserver
```

Open a browser tab to the room page at http://127.0.0.1:8000/chat/lobby/. Open a second browser tab to the same room page.

In the second browser tab, type the message "hello" and press enter. You should now see "hello" echoed in the chat log in both the second browser tab and in the first browser tab.

You now have a basic fully-functional chat server!

This tutorial continues in *Tutorial 3*.

2.3.3 Tutorial Part 3: Rewrite Chat Server as Asynchronous

This tutorial begins where *Tutorial 2* left off. We'll rewrite the consumer code to be asynchronous rather than synchronous to improve its performance.

Rewrite the consumer to be asynchronous

The `ChatConsumer` that we have written is currently synchronous. Synchronous consumers are convenient because they can call regular synchronous I/O functions such as those that access Django models without writing special code. However asynchronous consumers can provide a higher level of performance since they don't need to create additional threads when handling requests.

`ChatConsumer` only uses async-native libraries (Channels and the channel layer) and in particular it does not access synchronous Django models. Therefore it can be rewritten to be asynchronous without complications.

Note: Even if `ChatConsumer` *did* access Django models or other synchronous code it would still be possible to rewrite it as asynchronous. Utilities like *asgiref.sync.sync_to_async* and *channels.db.database_sync_to_async* can be used to call synchronous code from an asynchronous consumer. The performance gains however would be less than if it only used async-native libraries.

Let's rewrite `ChatConsumer` to be asynchronous. Put the following code in `chat/consumers.py`:

```
# chat/consumers.py
import json

from channels.generic.websocket import AsyncWebsocketConsumer


class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope["url_route"]["kwargs"]["room_name"]
        self.room_group_name = f"chat_{self.room_name}"

        # Join room group
        await self.channel_layer.group_add(self.room_group_name, self.channel_name)

        await self.accept()

    async def disconnect(self, close_code):
        # Leave room group
        await self.channel_layer.group_discard(self.room_group_name, self.channel_
↪name)

    # Receive message from WebSocket
    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]

        # Send message to room group
        await self.channel_layer.group_send(
            self.room_group_name, {"type": "chat.message", "message": message}
        )

    # Receive message from room group
    async def chat_message(self, event):
```

(continues on next page)

(continued from previous page)

```
        message = event["message"]

        # Send message to WebSocket
        await self.send(text_data=json.dumps({"message": message}))
```

This new code is for ChatConsumer is very similar to the original code, with the following differences:

- `ChatConsumer` now inherits from `AsyncWebsocketConsumer` rather than `WebsocketConsumer`.
- All methods are `async def` rather than just `def`.
- `await` is used to call asynchronous functions that perform I/O.
- `async_to_sync` is no longer needed when calling methods on the channel layer.

Let's verify that the consumer for the `/ws/chat/ROOM_NAME/` path still works. To start the Channels development server, run the following command:

```
$ python3 manage.py runserver
```

Open a browser tab to the room page at http://127.0.0.1:8000/chat/lobby/. Open a second browser tab to the same room page.

In the second browser tab, type the message "hello" and press enter. You should now see "hello" echoed in the chat log in both the second browser tab and in the first browser tab.

Now your chat server is fully asynchronous!

This tutorial continues in *Tutorial 4*.

2.3.4 Tutorial Part 4: Automated Testing

This tutorial begins where *Tutorial 3* left off. We've built a simple chat server and now we'll create some automated tests for it.

Testing the views

To ensure that the chat server keeps working, we will write some tests.

We will write a suite of end-to-end tests using Selenium to control a Chrome web browser. These tests will ensure that:

- when a chat message is posted then it is seen by everyone in the same room
- when a chat message is posted then it is not seen by anyone in a different room

Install the Chrome web browser, if you do not already have it.

Install chromedriver.

Install Selenium. Run the following command:

```
$ python3 -m pip install selenium
```

Create a new file `chat/tests.py`. Your app directory should now look like:

```
chat/
    __init__.py
    consumers.py
    routing.py
    templates/
        chat/
            index.html
            room.html
    tests.py
    urls.py
    views.py
```

Put the following code in `chat/tests.py`:

```
# chat/tests.py
from channels.testing import ChannelsLiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.wait import WebDriverWait


class ChatTests(ChannelsLiveServerTestCase):
    serve_static = True  # emulate StaticLiveServerTestCase

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        try:
            # NOTE: Requires "chromedriver" binary to be installed in $PATH
            cls.driver = webdriver.Chrome()
        except:
            super().tearDownClass()
            raise

    @classmethod
    def tearDownClass(cls):
        cls.driver.quit()
        super().tearDownClass()

    def test_when_chat_message_posted_then_seen_by_everyone_in_same_room(self):
        try:
            self._enter_chat_room("room_1")

            self._open_new_window()
            self._enter_chat_room("room_1")

            self._switch_to_window(0)
            self._post_message("hello")
            WebDriverWait(self.driver, 2).until(
                lambda _: "hello" in self._chat_log_value,
                "Message was not received by window 1 from window 1",
            )
            self._switch_to_window(1)
            WebDriverWait(self.driver, 2).until(
                lambda _: "hello" in self._chat_log_value,
```

(continues on next page)

(continued from previous page)

```
                "Message was not received by window 2 from window 1",
            )
        finally:
            self._close_all_new_windows()

    def test_when_chat_message_posted_then_not_seen_by_anyone_in_different_room(self):
        try:
            self._enter_chat_room("room_1")

            self._open_new_window()
            self._enter_chat_room("room_2")

            self._switch_to_window(0)
            self._post_message("hello")
            WebDriverWait(self.driver, 2).until(
                lambda _: "hello" in self._chat_log_value,
                "Message was not received by window 1 from window 1",
            )

            self._switch_to_window(1)
            self._post_message("world")
            WebDriverWait(self.driver, 2).until(
                lambda _: "world" in self._chat_log_value,
                "Message was not received by window 2 from window 2",
            )
            self.assertTrue(
                "hello" not in self._chat_log_value,
                "Message was improperly received by window 2 from window 1",
            )
        finally:
            self._close_all_new_windows()

    # === Utility ===

    def _enter_chat_room(self, room_name):
        self.driver.get(self.live_server_url + "/chat/")
        ActionChains(self.driver).send_keys(room_name, Keys.ENTER).perform()
        WebDriverWait(self.driver, 2).until(
            lambda _: room_name in self.driver.current_url
        )

    def _open_new_window(self):
        self.driver.execute_script('window.open("about:blank", "_blank");')
        self._switch_to_window(-1)

    def _close_all_new_windows(self):
        while len(self.driver.window_handles) > 1:
            self._switch_to_window(-1)
            self.driver.execute_script("window.close();")
        if len(self.driver.window_handles) == 1:
            self._switch_to_window(0)

    def _switch_to_window(self, window_index):
        self.driver.switch_to.window(self.driver.window_handles[window_index])

    def _post_message(self, message):
        ActionChains(self.driver).send_keys(message, Keys.ENTER).perform()
```

(continues on next page)

(continued from previous page)

```
    @property
    def _chat_log_value(self):
        return self.driver.find_element(
            by=By.CSS_SELECTOR, value="#chat-log"
        ).get_property("value")
```

Our test suite extends `ChannelsLiveServerTestCase` rather than Django's usual suites for end-to-end tests (`StaticLiveServerTestCase` or `LiveServerTestCase`) so that URLs inside the Channels routing configuration like `/ws/room/ROOM_NAME/` will work inside the suite.

We are using `sqlite3`, which for testing, is run as an in-memory database, and therefore, the tests will not run correctly. We need to tell our project that the `sqlite3` database need not to be in memory for run the tests. Edit the `mysite/settings.py` file and add the `TEST` argument to the **DATABASES** setting:

```
# mysite/settings.py
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.sqlite3",
        "NAME": BASE_DIR / "db.sqlite3",
        "TEST": {
            "NAME": BASE_DIR / "db.sqlite3",
        },
    }
}
```

To run the tests, run the following command:

```
$ python3 manage.py test chat.tests
```

You should see output that looks like:

```
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 5.014s

OK
Destroying test database for alias 'default'...
```

You now have a tested chat server!

What's next?

Congratulations! You've fully implemented a chat server, made it performant by writing it in asynchronous style, and written automated tests to ensure it won't break.

This is the end of the tutorial. At this point you should know enough to start an app of your own that uses Channels and start fooling around. As you need to learn new tricks, come back to rest of the *documentation*.

2.4 Consumers

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

Consumers do a couple of things in particular:

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

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

2.4.1 Basic Layout

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

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

```
from channels.consumer import SyncConsumer

class EchoConsumer(SyncConsumer):

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

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

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

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

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

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

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

```
from channels.consumer import AsyncConsumer

class EchoConsumer(AsyncConsumer):

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

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

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

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

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

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

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

Closing Consumers

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

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

The generic consumers below do this for you, so this is only needed if you are writing your own consumer class based on `AsyncConsumer` or `SyncConsumer`. However, if you override their `__call__` method, or block the handling methods that it calls from returning, you may still run into this; take a look at their source code if you want more information.

Additionally, if you launch your own background coroutines, make sure to also shut them down when the connection is finished, or you'll leak coroutines into the server.

Channel Layers

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

Consumers will use the channel layer `default` unless the `channel_layer_alias` attribute is set when subclassing any of the provided `Consumer` classes.

To use the channel layer `echo_alias` we would set it like so:

```
from channels.consumer import SyncConsumer

class EchoConsumer(SyncConsumer):
    channel_layer_alias = "echo_alias"
```

You can read more in *Channel Layers*.

2.4.2 Scope

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

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

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

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

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

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

2.4.3 Generic Consumers

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

WebsocketConsumer

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

```
from channels.generic.websocket import WebsocketConsumer

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

(continues on next page)

(continued from previous page)

```
    def connect(self):
        # Called on connection.
        # To accept the connection call:
        self.accept()
        # Or accept the connection and specify a chosen subprotocol.
        # A list of subprotocols specified by the connecting client
        # will be available in self.scope['subprotocols']
        self.accept("subprotocol")
        # To reject the connection, call:
        self.close()

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

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

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

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

AsyncWebsocketConsumer

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

```
from channels.generic.websocket import AsyncWebsocketConsumer

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

    async def connect(self):
        # Called on connection.
        # To accept the connection call:
        await self.accept()
        # Or accept the connection and specify a chosen subprotocol.
        # A list of subprotocols specified by the connecting client
        # will be available in self.scope['subprotocols']
```

(continues on next page)

(continued from previous page)

```
        await self.accept("subprotocol")
        # To reject the connection, call:
        await self.close()

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

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

JsonWebsocketConsumer

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

The only API differences are:

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

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

AsyncJsonWebsocketConsumer

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

AsyncHttpConsumer

Available as `channels.generic.http.AsyncHttpConsumer`, this offers basic primitives to implement a HTTP endpoint:

```
from channels.generic.http import AsyncHttpConsumer

class BasicHttpConsumer(AsyncHttpConsumer):
    async def handle(self, body):
        await asyncio.sleep(10)
        await self.send_response(200, b"Your response bytes", headers=[
            (b"Content-Type", b"text/plain"),
        ])
```

You are expected to implement your own `handle` method. The method receives the whole request body as a single bytestring. Headers may either be passed as a list of tuples or as a dictionary. The response body content is expected to be a bytestring.

You can also implement a `disconnect` method if you want to run code on disconnect - for example, to shut down any coroutines you launched. This will run even on an unclean disconnection, so don't expect that `handle` has finished running cleanly.

If you need more control over the response, e.g. for implementing long polling, use the lower level `self.send_headers` and `self.send_body` methods instead. This example already mentions channel layers which will be explained in detail later:

```
import json
from channels.generic.http import AsyncHttpConsumer

class LongPollConsumer(AsyncHttpConsumer):
    async def handle(self, body):
        await self.send_headers(headers=[
            (b"Content-Type", b"application/json"),
        ])
        # Headers are only sent after the first body event.
        # Set "more_body" to tell the interface server to not
        # finish the response yet:
        await self.send_body(b"", more_body=True)

    async def chat_message(self, event):
        # Send JSON and finish the response:
        await self.send_body(json.dumps(event).encode("utf-8"))
```

Of course you can also use those primitives to implement a HTTP endpoint for Server-sent events:

```
from datetime import datetime
from channels.generic.http import AsyncHttpConsumer

class ServerSentEventsConsumer(AsyncHttpConsumer):
    async def handle(self, body):
        await self.send_headers(headers=[
            (b"Cache-Control", b"no-cache"),
            (b"Content-Type", b"text/event-stream"),
            (b"Transfer-Encoding", b"chunked"),
        ])
        while True:
            payload = "data: %s\n\n" % datetime.now().isoformat()
            await self.send_body(payload.encode("utf-8"), more_body=True)
            await asyncio.sleep(1)
```

2.5 Routing

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

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

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

nest other, more protocol-specific routing underneath there.

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

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

```
import os

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application
from django.urls import path

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

from chat.consumers import AdminChatConsumer, PublicChatConsumer

application = ProtocolTypeRouter({
    # Django's ASGI application to handle traditional HTTP requests
    "http": django_asgi_app,

    # WebSocket chat handler
    "websocket": AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                path("chat/admin/", AdminChatConsumer.as_asgi()),
                path("chat/", PublicChatConsumer.as_asgi()),
            ])
        )
    ),
})
```

Note: We call the `as_asgi()` classmethod when routing our consumers. This returns an ASGI wrapper application that will instantiate a new consumer instance for each connection or scope. This is similar to Django's `as_view()`, which plays the same role for per-request instances of class-based views.

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

2.5.1 ProtocolTypeRouter

`channels.routing.ProtocolTypeRouter`

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

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

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

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

If you want to split HTTP handling between long-poll handlers and Django views, use a URLRouter using Django's `get_asgi_application()` specified as the last entry with a match-everything pattern.

2.5.2 URLRouter

`channels.routing.URLRouter`

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

```
URLRouter([
    re_path(r"^longpoll/$", LongPollConsumer.as_asgi()),
    re_path(r"^notifications/(?P<stream>\w+)/$", LongPollConsumer.as_asgi()),
    re_path(r"", get_asgi_application()),
])
```

Any captured groups will be provided in `scope` as the key `url_route`, a dict with a `kwargs` key containing a dict of the named regex groups and an `args` key with a list of positional regex groups. Note that named and unnamed groups cannot be mixed: Positional groups are discarded as soon as a single named group is matched.

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

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

Please note that `URLRouter` nesting will not work properly with `path()` routes if inner routers are wrapped by additional middleware. See Issue #1428.

2.5.3 ChannelNameRouter

`channels.routing.ChannelNameRouter`

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

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

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

2.6 Database Access

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

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

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

2.6.1 Database Connections

Channels can potentially open a lot more database connections than you may be used to if you are using threaded consumers (synchronous ones) - it can open up to one connection per thread.

If you wish to control the maximum number of threads used, set the `ASGI_THREADS` environment variable to the maximum number you wish to allow. By default, the number of threads is set to "the number of CPUs * 5" for Python 3.7 and below, and *min(32, os.cpu_count() + 4)* for Python 3.8+.

To avoid having too many threads idling in connections, you can instead rewrite your code to use async consumers and only dip into threads when you need to use Django's ORM (using `database_sync_to_async`).

2.6.2 database_sync_to_async

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

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

```
from channels.db import database_sync_to_async

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

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

You can also use it as a decorator:

```
from channels.db import database_sync_to_async

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

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

2.7 Channel Layers

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

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

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

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

2.7.1 Configuration

Channel layers are configured via the `CHANNEL_LAYERS` Django setting.

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

Redis Channel Layer

channels_redis is the only official Django-maintained channel layer supported for production use. The layer uses Redis as its backing store, and it supports both a single-server and sharded configurations as well as group support. To use this layer you'll need to install the channels_redis package.

In this example, Redis is running on localhost (127.0.0.1) port 6379:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```

In-Memory Channel Layer

Channels also comes packaged with an in-memory Channels Layer. This layer can be helpful in *Testing* or for local-development purposes:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels.layers.InMemoryChannelLayer"
    }
}
```

Warning: Do Not Use In Production

In-memory channel layers operate with each process as a separate layer. This means that no cross-process messaging is possible. As the core value of channel layers is to provide distributed messaging, in-memory usage will result in sub-optimal performance, and ultimately data-loss in a multi-instance environment.

2.7.2 Synchronous Functions

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

```
from asgiref.sync import async_to_sync

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

2.7.3 What To Send Over The Channel Layer

The channel layer is for high-level application-to-application communication. When you send a message, it is received by the consumers listening to the group or channel on the other end. What this means is that you should send high-level events over the channel layer, and then have consumers handle those events, and do appropriate low-level networking to their attached client.

For example, a chat application could send events like this over the channel layer:

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

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

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

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

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

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

they must all be synchronous (`def`).

2.7.4 Single Channels

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

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

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

```
class ChatConsumer(WebsocketConsumer):

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

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

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

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

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

```
from channels.layers import get_channel_layer

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

2.7.5 Groups

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

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

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

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

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

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

class ChatConsumer(WebsocketConsumer):

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

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

Note: Group names are restricted to ASCII alphanumerics, hyphens, and periods only and are limited to a maximum length of 100 in the default backend.

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

```
class ChatConsumer(WebsocketConsumer):

    ...

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

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

2.7.6 Using Outside Of Consumers

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

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

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

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

Or you'll need to use async_to_sync:

```
from asgiref.sync import async_to_sync

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

2.8 Sessions

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

2.8.1 Basic Usage

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

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

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

```
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from channels.sessions import SessionMiddlewareStack

from myapp import consumers

application = ProtocolTypeRouter({

    "websocket": AllowedHostsOriginValidator(
        SessionMiddlewareStack(
            URLRouter([
                path("frontend/", consumers.AsyncChatConsumer.as_asgi()),
            ])
        )
    ),

})
```

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

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

```
class ChatConsumer(WebsocketConsumer):

```

(continues on next page)

(continued from previous page)

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

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

2.8.2 Session Persistence

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

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

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

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

2.9 Authentication

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

2.9.1 Django authentication

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

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

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

```
from django.urls import re_path

from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from channels.security.websocket import AllowedHostsOriginValidator

from myapp import consumers

application = ProtocolTypeRouter({

    "websocket": AllowedHostsOriginValidator(
```

(continues on next page)

(continued from previous page)

```
        AuthMiddlewareStack(
            URLRouter([
                re_path(r"^front(end)/$", consumers.AsyncChatConsumer.as_asgi()),
            ])
        )
    ),

})
```

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

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

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

```
class ChatConsumer(WebsocketConsumer):

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

2.9.2 Custom Authentication

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

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

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

```
from channels.db import database_sync_to_async

@database_sync_to_async
def get_user(user_id):
    try:
        return User.objects.get(id=user_id)
    except User.DoesNotExist:
        return AnonymousUser()

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

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

    async def __call__(self, scope, receive, send):
        # Look up user from query string (you should also do things like
        # checking if it is a valid user ID, or if scope["user"] is already
        # populated).
        scope['user'] = await get_user(int(scope["query_string"]))
```

(continues on next page)

(continued from previous page)

```
        return await self.app(scope, receive, send)
```

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

2.9.3 How to log a user in/out

Channels provides direct login and logout functions (much like Django's `contrib.auth` package does) as `channels.auth.login` and `channels.auth.logout`.

Within your consumer you can await `login(scope, user, backend=None)` to log a user in. This requires that your scope has a `session` object; the best way to do this is to ensure your consumer is wrapped in a `SessionMiddlewareStack` or a `AuthMiddlewareStack`.

You can logout a user with the `logout(scope)` async function.

If you are in a WebSocket consumer, or logging-in after the first response has been sent in a http consumer, the session is populated **but will not be saved automatically** - you must call `scope["session"].save()` after login in your consumer code:

```
from channels.auth import login

class ChatConsumer(AsyncWebsocketConsumer):

    ...

    async def receive(self, text_data):
        ...
        # login the user to this session.
        await login(self.scope, user)
        # save the session (if the session backend does not access the db you can use␣
↪`sync_to_async`)
        await database_sync_to_async(self.scope["session"].save)()
```

When calling `login(scope, user)`, `logout(scope)` or `get_user(scope)` from a synchronous function you will need to wrap them in `async_to_sync`, as we only provide async versions:

```
from asgiref.sync import async_to_sync
from channels.auth import login

class SyncChatConsumer(WebsocketConsumer):

    ...

    def receive(self, text_data):
        ...
        async_to_sync(login)(self.scope, user)
        self.scope["session"].save()
```

Note: If you are using a long running consumer, websocket or long-polling HTTP it is possible that the user will be logged out of their session elsewhere while your consumer is running. You can periodically use `get_user(scope)` to be sure that the user is still logged in.

2.10 Security

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

2.10.1 WebSockets

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

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

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

`OriginValidator` lets you restrict the valid options for the `Origin` header that is sent with every WebSocket to say where it comes from. Just wrap it around your WebSocket application code like this, and pass it a list of valid domains as the second argument. You can pass only a single domain (for example, `.allowed-domain.com`) or a full origin, in the format `scheme://domain[:port]` (for example, `http://allowed-domain.com:80`). Port is optional, but recommended:

```
from channels.security.websocket import OriginValidator

application = ProtocolTypeRouter({

    "websocket": OriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                ...
            ])
        ),
        [".goodsite.com", "http://.goodsite.com:80", "http://other.site.com"],
    ),
})
```

Note: If you want to resolve any domain, then use the origin `*`.

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

```
from channels.security.websocket import AllowedHostsOriginValidator

application = ProtocolTypeRouter({

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

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

2.11 Testing

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

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

You can test asynchronous code using Django's `TestCase`. Alternately, you can use `pytest` with its pytest-asyncio plugin.

2.11.1 Setting Up Async Tests

To use Django's `TestCase` you simply define an `async def` test method in order to provide the appropriate async context:

```
from django.test import TestCase
from channels.testing import HttpCommunicator
from myproject.myapp.consumers import MyConsumer

class MyTests(TestCase):
    async def test_my_consumer(self):
        communicator = HttpCommunicator(MyConsumer.as_asgi(), "GET", "/test/")
        response = await communicator.get_response()
        self.assertEqual(response["body"], b"test response")
        self.assertEqual(response["status"], 200)
```

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

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

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

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

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

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

2.11.2 ApplicationCommunicator

`ApplicationCommunicator` is the generic test helper for any ASGI application. It provides several basic methods for interaction as explained below.

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

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

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

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

send_input

Call it to send an event to the application:

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

receive_output

Call it to receive an event from the application:

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

receive_nothing

Call it to check that there is no event waiting to be received from the application:

```
assert await communicator.receive_nothing(timeout=0.1, interval=0.01) is False
# Receive the rest of the http request from above
event = await communicator.receive_output()
assert event["type"] == "http.response.body"
assert event.get("more_body") is True
event = await communicator.receive_output()
assert event["type"] == "http.response.body"
assert event.get("more_body") is None
# Check that there isn't another event
assert await communicator.receive_nothing() is True
# You could continue to send and receive events
# await communicator.send_input(...)
```

The method has two optional parameters:

- `timeout`: number of seconds to wait to ensure the queue is empty. Defaults to 0.1.
- `interval`: number of seconds to wait for another check for new events. Defaults to 0.01.

wait

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

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

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

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

2.11.3 HttpCommunicator

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

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

And then wait for its response:

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

You can pass the following arguments to the constructor:

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

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

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

2.11.4 WebsocketCommunicator

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

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

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

Important: If you don't call `WebsocketCommunicator.disconnect()` before your test suite ends, you may find yourself getting `RuntimeWarnings` about things never being awaited, as you will be killing your app off in the middle of its lifecycle. You do not, however, have to `disconnect()` if your app already raised an error.

You can also pass an `application` built with `URLRouter` instead of the plain consumer class. This lets you test applications that require positional or keyword arguments in the `scope`:

```
from channels.testing import WebsocketCommunicator
application = URLRouter([
    path("testws/<message>/", KwargsWebSocketApp.as_asgi()),
])
communicator = WebsocketCommunicator(application, "/testws/test/")
connected, subprotocol = await communicator.connect()
assert connected
# Test on connection welcome message
message = await communicator.receive_from()
assert message == 'test'
# Close
await communicator.disconnect()
```

Note: Since the `WebsocketCommunicator` class takes a URL in its constructor, a single Communicator can only test a single URL. If you want to test multiple different URLs, use multiple Communicators.

connect

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

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

send_to

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

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

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

send_json_to

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

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

receive_from

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

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

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

receive_json_from

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

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

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

receive_nothing

Checks that there is no frame waiting to be received from the application. For details see *ApplicationCommunicator*.

disconnect

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

You do not need to call this if the application instance you're testing already exited (for example, if it errored), but if you do call it, it will just silently return control to you.

2.11.5 ChannelsLiveServerTestCase

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

```
from channels.testing import ChannelsLiveServerTestCase

class SomeLiveTests(ChannelsLiveServerTestCase):

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

Note: You can't use an in-memory database for your live tests. Therefore include a test database file name in your settings to tell Django to use a file database if you use SQLite:

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

serve_static

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

2.12 Worker and Background Tasks

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

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

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

This feature does not work with the in-memory channel layer.

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

2.12.1 Sending

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

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

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

Also remember that if you are sending the event from a synchronous environment, you have to use the `asgiref.sync.async_to_sync` wrapper as specified in *channel layers*.

2.12.2 Receiving and Consumers

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

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

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

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

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

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

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

2.13 Deploying

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

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

2.13.1 Configuring the ASGI application

As discussed in *Installation* and *Routing*, you will have a file like `myproject/asgi.py` that will define your *root application*. This is almost certainly going to be your top-level (Protocol Type) router.

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

```
import os

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

(continues on next page)

(continued from previous page)

```
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application
from django.urls import path

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

from chat.consumers import AdminChatConsumer, PublicChatConsumer

application = ProtocolTypeRouter({
    # Django's ASGI application to handle traditional HTTP requests
    "http": django_asgi_app,

    # WebSocket chat handler
    "websocket": AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                path("chat/admin/", AdminChatConsumer.as_asgi()),
                path("chat/", PublicChatConsumer.as_asgi()),
            ])
        )
    ),
})
```

2.13.2 Setting up a channel backend

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

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

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

To use the Redis backend you have to install it:

```
pip install -U channels_redis
```

2.13.3 Run protocol servers

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

All these servers have their own configuration options, but they all have one thing in common - they will want you to pass them an ASGI application to run. All you need to do is pass the `application` object inside your project's `asgi.py` file to your protocol server as the application it should run:

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

2.13.4 HTTP and WebSocket

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

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

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

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

To run Daphne, it just needs to be supplied with an application, much like a WSGI server would need to be. Make sure you have an `asgi.py` file as outlined above.

Then, you can run Daphne and supply the ASGI application as the argument:

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

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

If you want to bind multiple Daphne instances to the same port on a machine, use a process supervisor that can listen on ports and pass the file descriptors to launched processes, and then pass the file descriptor with `--fd NUM`.

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

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

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

2.13.5 Alternative Web Servers

There are also alternative ASGI servers that you can use for serving Channels.

To some degree ASGI web servers should be interchangeable, they should all have the same basic functionality in terms of serving HTTP and WebSocket requests.

Aspects where servers may differ are in their configuration and defaults, performance characteristics, support for resource limiting, differing protocol and socket support, and approaches to process management.

You can see more alternative servers, such as Uvicorn, in the ASGI implementations documentation.

2.13.6 Example Setups

These are examples of possible setups - they are not guaranteed to work out of the box, and should be taken more as a guide than a direct tutorial.

Nginx/Supervisor (Ubuntu)

This example sets up a Django site on an Ubuntu server, using Nginx as the main webserver and supervisord to run and manage Daphne.

First, install Nginx and Supervisor:

```
$ sudo apt install nginx supervisor
```

Now, you will need to create the supervisor configuration file (often located in `/etc/supervisor/conf.d/` - here, we're making Supervisor listen on the TCP port and then handing that socket off to the child processes so they can all share the same bound port:

```
[fcgi-program:asgi]
# TCP socket used by Nginx backend upstream
socket=tcp://localhost:8000

# Directory where your site's project files are located
directory=/my/app/path

# Each process needs to have a separate socket file, so we use process_num
# Make sure to update "mysite.asgi" to match your project name
command=daphne -u /run/daphne/daphne%(process_num)d.sock --fd 0 --access-log - --
↪proxy-headers mysite.asgi:application

# Number of processes to startup, roughly the number of CPUs you have
numprocs=4

# Give each process a unique name so they can be told apart
process_name=asgi%(process_num)d

# Automatically start and recover processes
autostart=true
autorestart=true

# Choose where you want your log to go
stdout_logfile=/your/log/asgi.log
redirect_stderr=true
```

Create the run directory for the sockets referenced in the supervisor configuration file.

```
$ sudo mkdir /run/daphne/
```

When running the supervisor fcgi-program under a different user, change the owner settings of the run directory.

```
$ sudo chown <user>.<group> /run/daphne/
```

The /run/ folder is cleared on a server reboot. To make the /run/daphne folder persistant create a file `/usr/lib/tmpfiles.d/daphne.conf` with the contents below.

```
$ d /run/daphne 0755 <user> <group>
```

Have supervisor reread and update its jobs:

```
$ sudo supervisorctl reread
$ sudo supervisorctl update
```

Note: Running the daphne command with `--fd 0` in the commandline will fail and result in *[Errno 88] Socket operation on non-socket*.

Supervisor will automatically create the socket, bind, and listen before forking the first child in a group. The socket will be passed to each child on file descriptor number 0 (zero). See https://supervisord.org/configuration.html#fcgi-program-x-section-settings

Next, Nginx has to be told to proxy traffic to the running Daphne instances. Setup your nginx upstream conf file for your project:

```
upstream channels-backend {
    server localhost:8000;
}
...
server {
    ...
    location / {
        try_files $uri @proxy_to_app;
    }
    ...
    location @proxy_to_app {
        proxy_pass http://channels-backend;

        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $server_name;
    }
    ...
}
```

Reload nginx to apply the changes:

```
$ sudo service nginx reload
```

2.14 Troubleshooting

2.14.1 ImproperlyConfigured exception

```
django.core.exceptions.ImproperlyConfigured: Requested setting INSTALLED_APPS, but␣
↪settings are not configured.
You must either define the environment variable DJANGO_SETTINGS_MODULE or call␣
↪settings.configure() before accessing settings.
```

This exception occurs when your application tries to import any models before Django finishes its initialization process aka `django.setup()`.

`django.setup()` should be called only once, and should be called manually only in case of standalone apps. In context of Channels usage, `django.setup()` is called automatically in `get_asgi_application()`, which means it needs to be called before any ORM models are imported.

The working code order would look like this:

```
import os

from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.core.asgi import get_asgi_application
from django.urls import path

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
# Initialize Django ASGI application early to ensure the AppRegistry
# is populated before importing code that may import ORM models.
django_asgi_app = get_asgi_application()

from chat.consumers import AdminChatConsumer, PublicChatConsumer

application = ProtocolTypeRouter({
    # Django's ASGI application to handle traditional HTTP requests
    "http": django_asgi_app,

    # WebSocket chat handler
    "websocket": AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                path("chat/admin/", AdminChatConsumer.as_asgi()),
                path("chat/", PublicChatConsumer.as_asgi()),
            ])
        )
    ),
})
```

CHAPTER 3

Reference

3.1 ASGI

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

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

The full ASGI spec can be found at https://asgi.readthedocs.io

3.1.1 Summary

ASGI is structured as a single asynchronous callable, which takes a dict `scope` and two callables `receive` and `send`:

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

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

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

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

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

3.1.2 Composability

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

3.1.3 Protocol Specifications

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

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

3.2 Channel Layer Specification

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

3.2.1 Abstract

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

3.2.2 Overview

Messages

Messages must be a `dict`. Because these messages are sometimes sent over a network, they need to be serializable, and so they are only allowed to contain the following types:

- Byte strings
- Unicode strings
- Integers (within the signed 64 bit range)
- Floating point numbers (within the IEEE 754 double precision range)
- Lists (tuples should be encoded as lists)
- Dicts (keys must be unicode strings)
- Booleans
- None

Channels

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

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

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

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

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

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

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

Extensions

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

The extensions defined here are:

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

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

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

Asynchronous Support

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

Groups

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

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

Capacity

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

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

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

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

3.2.3 Specification Details

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

- `coroutine send(channel, message)`, that takes two arguments: the channel to send on, as a unicode string, and the message to send, as a serializable `dict`.
- `coroutine receive(channel)`, that takes a single channel name and returns the next received message on that channel.
- `coroutine new_channel()`, which returns a new process-specific channel that can be used to give to a local coroutine or receiver.
- `MessageTooLarge`, the exception raised when a send operation fails because the encoded message is over the layer's size limit.
- `ChannelFull`, the exception raised when a send operation fails because the destination channel is over capacity.
- `extensions`, a list of unicode string names indicating which extensions this layer provides, or an empty list if it supports none. The possible extensions can be seen in *Extensions*.

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

- `coroutine group_add(group, channel)`, that takes a `channel` and adds it to the group given by `group`. Both are unicode strings. If the channel is already in the group, the function should return normally.
- `coroutine group_discard(group, channel)`, that removes the `channel` from the `group` if it is in it, and does nothing otherwise.

- `coroutine group_send(group, message)`, that takes two positional arguments; the group to send to, as a unicode string, and the message to send, as a serializable `dict`. It may raise MessageTooLarge but cannot raise ChannelFull.
- `group_expiry`, an integer number of seconds that specifies how long group membership is valid for after the most recent `group_add` call (see *Persistence* below)

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

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

Channel Semantics

Channels **must**:

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

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

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

Persistence

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

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

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

Approximate Global Ordering

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

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

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

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

Strings and Unicode

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

Unicode string refers to `unicode` on Python 2 and `str` on Python 3. This document will never specify just *string* - all strings are one of the two exact types.

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

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

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

3.2.4 Copyright

3.3 Community Projects

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

- Beatserver, a periodic task scheduler for Django Channels.
- EventStream, a library to push data using the Server-Sent Events (SSE) protocol.
- DjangoChannelsRestFramework, a framework that provides DRF-like consumers for Channels.
- ChannelsMultiplexer, a JsonConsumer Multiplexer for Channels.
- DjangoChannelsIRC, an interface server and matching generic consumers for IRC.
- Apollo, a real-time polling application for corporate and academic environments.
- DjangoChannelsJsonRpc, a wrapper for the JSON-RPC protocol.

- channels-demultiplexer, a (de)multiplexer for `AsyncJsonWebsocketConsumer` consumers.
- channels_postgres, a Django Channels channel layer that uses PostgreSQL as its backing store.
- channels-auth-token-middlewares, Django REST framework token authentication middleware and SimpleJWT middleware, such as QueryStringSimpleJWTAuthTokenMiddleware for WebSocket authentication.

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

3.4 Contributing

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

3.4.1 What can I work on?

We're looking for help with the following areas:

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

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

- Channels issues, for the Django integration and overall project efforts
- Daphne issues, for the HTTP and Websocket termination
- asgiref issues, for the base ASGI library/memory backend
- channels_redis issues, for the Redis channel backend

Issues are categorized by difficulty level:

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

They are also classified by type:

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

You should filter the issues list by the experience level and type of work you'd like to do, and then if you want to take something on leave a comment and assign yourself to it. If you want advice about how to take on a bug, leave a comment asking about it and we'll be happy to help.

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

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

Of course! The issues labelled with `exp/beginner` are a perfect place to get started, as they're usually small and well defined. If you want help with one of them, jump in and comment on the ticket if you need input or assistance.

3.4.3 How do I get started and run the tests?

First, you should first clone the git repository to a local directory:

```
git clone https://github.com/django/channels.git channels
```

Next, you may want to make a virtual environment to run the tests and develop in; you can use either `virtualenvwrapper`, `pipenv` or just plain `virtualenv` for this.

Then, `cd` into the `channels` directory and install it editable into your environment:

```
cd channels/
python -m pip install -e .[tests]
```

Note the `[tests]` section there; that tells `pip` that you want to install the `tests` extra, which will bring in testing dependencies like `pytest-django`.

Then, you can run the tests:

```
pytest
```

Also, there is a tox.ini file at the root of the repository. Example commands:

```
 $ tox -l
 py37-dj32
 py38-dj32
 py39-dj32
 py310-dj32
 py38-dj40
 py38-dj41
 py38-djmain
 py39-dj40
 py39-dj41
 py39-djmain
 py310-dj40
 py310-dj41
 py310-djmain
 qa

# run the test with Python 3.10, on Django 4.1 and Django main branch
$ tox -e py310-dj41,py310-djmain
```

Note that tox can also forward arguments to pytest. When using pdb with pytest, forward the `-s` option to pytest as such:

```
tox -e py310-dj41 -- -s
```

The `qa` environment runs the various linters used by the project.

3.4.4 How do I do a release?

If you have commit access, a release involves the following steps:

- Create a new entry in the CHANGELOG.txt file and summarise the changes
- Create a new release page in the docs under `docs/releases` and add the changelog there with more information where necessary
- Add a link to the new release notes in `docs/releases/index.rst`
- Set the new version in `__init__.py`
- Roll all of these up into a single commit and tag it with the new version number. Push the commit and tag.
- To upload you will need to be added as a maintainer on PyPI. Run *python setup.py sdist bdist_wheel*, and *twine upload*.

The release process for `channels-redis` and `daphne` is similar, but they don't have the two steps in `docs/`.

3.5 Support

If you have questions about Channels, need debugging help or technical support, you can turn to community resources like:

- Stack Overflow
- The Django Users mailing list (django-users@googlegroups.com)
- The #django channel on the PySlackers Slack group

If you have a concrete bug or feature request (one that is clear and actionable), please file an issue against the appropriate GitHub project.

Unfortunately, if you open a GitHub issue with a vague problem (like "it's slow!" or "connections randomly drop!") we'll have to close it as we don't have the volunteers to answer the number of questions we'd get - please go to one of the other places above for support from the community at large.

As a guideline, your issue is concrete enough to open an issue if you can provide **exact steps to reproduce** in a fresh, example project. We need to be able to reproduce it on a *normal, local developer machine* - so saying something doesn't work in a hosted environment is unfortunately not very useful to us, and we'll close the issue and point you here.

Apologies if this comes off as harsh, but please understand that open source maintenance and support takes up a lot of time, and if we answered all the issues and support requests there would be no time left to actually work on the code itself!

3.5.1 Making bugs reproducible

If you're struggling with an issue that only happens in a production environment and can't get it to reproduce locally so either you can fix it or someone can help you, take a step-by-step approach to eliminating the differences between the environments.

First off, try changing your production environment to see if that helps - for example, if you have Nginx/Apache/etc. between browsers and Channels, try going direct to the Python server and see if that fixes things. Turn SSL off if you have it on. Try from different browsers and internet connections. WebSockets are notoriously hard to debug already, and so you should expect some level of awkwardness from any project involving them.

Next, check package versions between your local and remote environments. You'd be surprised how easy it is to forget to upgrade something!

Once you've made sure it's none of that, try changing your project. Make a fresh Django project (or use one of the Channels example projects) and make sure it doesn't have the bug, then work on adding code to it from your project until the bug appears. Alternately, take your project and remove pieces back down to the basic Django level until it works.

Network programming is also just difficult in general; you should expect some level of reconnects and dropped connections as a matter of course. Make sure that what you're seeing isn't just normal for a production application.

3.5.2 How to help the Channels project

If you'd like to help us with support, the first thing to do is to provide support in the communities mentioned at the top (Stack Overflow and the mailing list).

If you'd also like to help triage issues, please get in touch and mention you'd like to help out and we can make sure you're set up and have a good idea of what to do. Most of the work is making sure incoming issues are actually valid and actionable, and closing those that aren't and redirecting them to this page politely and explaining why.

Some sample response templates are below.

General support request

```
Sorry, but we can't help out with general support requests here - the issue tracker␣
↪is for reproduceable bugs and
concrete feature requests only! Please see our support documentation (https://
↪channels.readthedocs.io/en/latest/support.html)
for more information about where you can get general help.
```

Non-specific bug/"It doesn't work!"

```
I'm afraid we can't address issues without either direct steps to reproduce, or that␣
↪only happen in a production
environment, as they may not be problems in the project itself. Our support␣
↪documentation
(https://channels.readthedocs.io/en/latest/support.html) has details about how to␣
↪take this sort of problem, diagnose it,
and either fix it yourself, get help from the community, or make it into an␣
↪actionable issue that we can handle.

Sorry we have to direct you away like this, but we get a lot of support requests␣
↪every week. If you can reduce the problem
to a clear set of steps to reproduce or an example project that fails in a fresh␣
↪environment, please re-open the ticket
with that information.
```

Problem in application code

```
It looks like a problem in your application code rather than in Channels itself, so I
↪'m going to close the ticket.
If you can trace it down to a problem in Channels itself (with exact steps to␣
↪reproduce on a fresh or small example
project - see https://channels.readthedocs.io/en/latest/support.html) please re-open␣
↪the ticket! Thanks.
```

3.6 Release Notes

3.6.1 4.0.0 Release Notes

Channels 4 is the next major version of the Channels package. Together with the matching Daphne v4 and channels-redis v4 releases, it updates dependencies, fixes issues, and removes outdated code. It so provides the foundation for Channels development going forward.

In most cases, you can update now by updating `channels`, `daphne`, and `channels-redis` as appropriate, with `pip`, and by adding `daphne` at the top of your `INSTALLED_APPS` setting.

First `pip`:

```
pip install -U 'channels[daphne]' channels-redis
```

Then in your Django settings file:

```
INSTALLED_APPS = [
    "daphne",
    ...
]
```

Read on for the details.

Updated Python and Django support

In general Channels will try to follow Python and Django supported versions.

As of release, that means Python 3.7, 3.8, 3.9, and 3.10, as well as Django 3.2, 4.0, and 4.1 are currently supported.

As a note, we reserve the right to drop older Python versions, or the older Django LTS, once the newer one is released, before their official end-of-life if this is necessary to ease development.

Dropping older Python and Django versions will be done in minor version releases, and will not be considered to require a major version change.

The async support in both Python and Django continues to evolve rapidly. We advise you to always upgrade to the latest versions in order to avoid issues in older versions if you're building an async application.

- Dropped support for Python 3.6.
- Minimum Django version is now Django 3.2.
- Added compatibility with Django 4.1.

Decoupling of the Daphne application server

In order to allow users of other ASGI servers to use Channels without the overhead of Daphne and Twisted, the Daphne application server is now an optional dependency, installable either directly or with the `daphne` extra, as per the `pip` example above.

- Where Daphne is used `daphne>=4.0.0` is required. The `channels[daphne]` extra assures this.
- The `runserver` command is moved to the `daphne` package.

  In order to use the `runserver` command, add `daphne` to your `INSTALLED_APPS`, before `django.contrib.staticfiles`:

  ```
  INSTALLED_APPS = [
      "daphne",
      ...
  ]
  ```

  There is a new system check to ensure this ordering.

  Note, the `runworker` command remains a part of the `channels` app.
- Use of `ChannelsLiveServerTestCase` still requires Daphne.

Removal of the Django application wrappers

In order to add initial ASGI support to Django, Channels originally provided tools for wrapping your Django application and serving it under ASGI. This included an ASGI handler class, an ASGI HTTP request object, and an ASGI compatible version of the staticfiles handler for use with `runserver`

Improved equivalents to all of these are what has been added to Django since Django version 3.0. As such serving of Django HTTP applications (whether using sync or async views) under ASGI is now Django's responsibility, and the matching Channels classes have been removed.

Use of these classes was deprecated in Channels v3 and, if you've already moved to the Django equivalents there is nothing further to do.

- Removed deprecated static files handling in favor of `django.contrib.staticfiles`.
- Removed the deprecated AsgiHandler, which wrapped Django views, in favour of Django's own ASGI support. You should use Django's `get_asgi_application` to provide the `http` handler for ProtocolTypeRouter, or an appropriate path for URLRouter, in order to route your Django application.
- The supporting `AsgiRequest` is also removed, as it was only used for `AsgiHandler`.
- Removed deprecated automatic routing of `http` protocol handler in `ProtocolTypeRouter`. You must explicitly register the `http` handler in your application if using `ProtocolTypeRouter`.

The minimal `asgi.py` file routing the Django ASGI application under a `ProtocolTypeRouter` will now look something like this:

```
import os

from channels.routing import ProtocolTypeRouter
from django.core.asgi import get_asgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'mysite.settings')

application = ProtocolTypeRouter({
```

(continues on next page)

(continued from previous page)

```
    "http": get_asgi_application(),
})
```

i.e. We use Django's `get_asgi_application()`, and explicitly route an `http` handler for `ProtocolTypeRouter`. This is merely for illustration of the changes. Please see the docs for more complete examples.

Other changes

- The use of the `guarantee_single_callable()` compatibility shim is removed. All applications must be ASGI v3 single-callables.
- Removed the `consumer_started` and `consumer_finished` signals, unused since the 2.0 rewrite.
- Fixed `ChannelsLiveServerTestCase` when running on systems using the `spawn` multiprocessing start method, such as macOS and Windows.

3.6.2 3.0.5 Release Notes

Channels 3.0.5 is a bugfix release in the 3.0 series.

Bugfixes & Small Changes

- Removed use of `providing_args` keyword argument to consumer started signal, as support for this was removed in Django 4.0.

Backwards Incompatible Changes

- Drops support for end-of-life Python 3.6 and Django 3.0 and 3.1.

3.6.3 3.0.4 Release Notes

Channels 3.0.4 is a bugfix release in the 3.0 series.

Bugfixes & Small Changes

- Usage of `urlparse` in `OriginValidator` is corrected to maintain compatibility with recent point-releases of Python.
- The import of `django.contrib.auth.models.AnonymousUser` in `channels.auth` is deferred until runtime, in order to avoid errors if `AuthMiddleware` or `AuthMiddlewareStack` were imported before `django.setup()` was run.
- `CookieMiddleware` adds support for the `samesite` flag.
- `WebsocketConsumer.init()` and `AsyncWebsocketConsumer.init()` no longer make a bad *super()* call to `object.init()`.

Backwards Incompatible Changes

None.

3.6.4 3.0.3 Release Notes

Channels 3.0.3 fixes a security issue in Channels 3.0.2

CVE-2020-35681: Potential leakage of session identifiers using legacy `AsgiHandler`

The legacy `channels.http.AsgiHandler` class, used for handling HTTP type requests in an ASGI environment prior to Django 3.0, did not correctly separate request scopes in Channels 3.0. In many cases this would result in a crash but, with correct timing responses could be sent to the wrong client, resulting in potential leakage of session identifiers and other sensitive data.

This issue affects Channels 3.0.x before 3.0.3, and is resolved in Channels 3.0.3.

Users of `ProtocolTypeRouter` not explicitly specifying the handler for the `'http'` key, or those explicitly using `channels.http.AsgiHandler`, likely to support Django v2.2, are affected and should update immediately.

Note that both an unspecified handler for the `'http'` key and using `channels.http.AsgiHandler` are deprecated, and will raise a warning, from Channels v3.0.0

This issue affects only the legacy channels provided class, and not Django's similar `ASGIHandler`, available from Django 3.0. It is recommended to update to Django 3.0+ and use the Django provided `ASGIHandler`.

A simplified `asgi.py` script will look like this:

```
import os

from django.core.asgi import get_asgi_application

# Fetch Django ASGI application early to ensure AppRegistry is populated
# before importing consumers and AuthMiddlewareStack that may import ORM
# models.
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "mysite.settings")
django_asgi_app = get_asgi_application()

# Import other Channels classes and consumers here.
from channels.routing import ProtocolTypeRouter, URLRouter

application = ProtocolTypeRouter({
    # Explicitly set 'http' key using Django's ASGI application.
    "http": django_asgi_app,
    ),
})
```

Please see *Deploying* for a more complete example.

3.6.5 3.0.2 Release Notes

Channels 3.0.2 fixes a bug in Channels 3.0.1

Bugfixes

- Fixes a bug in Channels 3.0 where *StaticFilesWrapper* was not updated to the ASGI 3 single-callable interface.
- Users of the `runworker` command should ensure to update `asgiref` to version 3.3.1 or later, where an issue in `asgiref.server.StatelessServer` was addressed.

3.6.6 3.0.1 Release Notes

Channels 3.0.1 fixes a bug in Channels 3.0.

Bugfixes

- Fixes a bug in Channels 3.0 where `SessionMiddleware` would not correctly isolate per-instance scopes.

3.6.7 3.0.0 Release Notes

The Channels 3 update brings Channels into line with Django's own async ASGI support, introduced with Django 3.0.

Channels now integrates with Django's async HTTP handling, whilst continuing to support WebSockets and other exciting consumer types.

Channels 3 supports Django 3.x and beyond, as well continuing to support the Django 2.2 LTS. We will support Django 2.2 at least until the Django 3.2 LTS is released, yet may drop support after that, but before Django 2.2 is officially end-of-life.

Likewise, we support Python 3.6+ but we **strongly advise** you to update to the latest Python versions, so 3.9 at the time of release.

In both our Django and Python support, we reflect the reality that async Python and async Django are still both evolving rapidly. Many issues we see simply disappear if you update. Whatever you are doing with async, you should make sure you're on the latest versions.

The highlight of this release is the upgrade to ASGI v3, which allows integration with Django's ASGI support. There are also two additional deprecations that you will need to deal with if you are updating an existing application.

Update to ASGI 3

- Consumers are now ASGI 3 *single-callables* with the signature:

```
application(scope, receive, send)
```

  For generic consumers this change should be largely transparent, but you will need to update `__init__()` (no longer taking the scope) and `__call__()` (now taking the scope) **if you implemented these yourself**.

- Consumers now have an `as_asgi()` class method you need to call when setting up your routing:

```
websocket_urlpatterns = [
    re_path(r'ws/chat/(?P<room_name>\w+)/$', consumers.ChatConsumer.as_asgi()),
]
```

  This returns an ASGI application that will instantiate the consumer per-request. It's similar to Django's `as_view()`, which serves the same purpose. You can pass in keyword arguments for initialization if your consumer requires them.

- Middleware will also need to be updated to the ASGI v3 signature. The `channels.middleware.BaseMiddleware` class is simplified, and available as an example. You probably don't need to actually subclass it under ASGI 3.

Deprecations

- Using `ProtocolTypeRouter` without an explicit `"http"` key is now deprecated.

  Following Django conventions, your entry point script should be named `asgi.py`, and you should use Django's `get_asgi_application()`, that is used by Django's default `asgi.py` template to route the `"http"` handler:

```
from django.core.asgi import get_asgi_application

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    # Other protocols here.
})
```

  Once the deprecation is removed, when we drop support for Django 2.2, not specifying an `"http"` key will mean that your application will not handle HTTP requests.

- The Channels built-in HTTP protocol `AsgiHandler` is also deprecated. You should update to Django 3.0 or higher and use Django's `get_asgi_application()`. Channel's `AsgiHandler` will be removed when we drop support for Django 2.2.

3.6.8 2.4.0 Release Notes

Channels 2.4 brings compatibility with Django 3.0s `async_unsafe()` checks. (Specifically we ensure session save calls are made inside an asgiref `database_sync_to_async()`.)

If you are using Daphne, it is recommended that you install Daphne version 2.4.1 or later for full compatibility with Django 3.0.

Backwards Incompatible Changes

In line with the guidance provided by Django's supported versions policy we now also drop support for all Django versions before 2.2, which is the current LTS.

3.6.9 2.3.0 Release Notes

Channels 2.3.0 updates the `AsgiHandler` HTTP request body handling to use a spooled temporary file, rather than reading the whole request body into memory.

This significantly reduces the maximum memory requirements when serving Django views, and protects from DoS attacks, whilst still allowing large file uploads — a combination that had previously been *difficult*.

Many thanks to Ivan Ergunov for his work on the improvements!

Backwards Incompatible Changes

As a result of the reworked body handling, `AsgiRequest.__init__()` is adjusted to expect a file-like `stream`, rather than the whole `body` as bytes.

Test cases instantiating requests directly will likely need to be updated to wrap the provided `body` in, e.g., `io.BytesIO`.

Next Up...

We're looking to address a few issues around `AsyncHttpConsumer`. Any human-power available to help on that, truly appreciated.

3.6.10 2.2.0 Release Notes

Channels 2.2.0 updates the requirements for ASGI version 3, and the supporting Daphne v2.3 release.

Backwards Incompatible Changes

None.

3.6.11 2.1.7 Release Notes

Channels 2.1.7 is another bugfix release in the 2.1 series, and the last release (at least for a long while) with Andrew Godwin as the primary maintainer.

Thanks to everyone who has used, supported, and contributed to Channels over the years, and I hope we can keep it going with community support for a good while longer.

Bugfixes & Small Changes

- HTTP request body size limit is now enforced (the one set by the `DATA_UPLOAD_MAX_MEMORY_SIZE` setting)
- `database_sync_to_async` now closes old connections before it runs code, which should prevent some connection errors in long-running pages or tests.
- The auth middleware closes old connections before it runs, to solve similar old-connection issues.

Backwards Incompatible Changes

None.

3.6.12 2.1.6 Release Notes

Channels 2.1.6 is another bugfix release in the 2.1 series.

Bugfixes & Small Changes

- HttpCommunicator now extracts query strings correctly from its provided arguments
- AsyncHttpConsumer provides channel layer attributes following the same conventions as other consumer classes
- Prevent late-Daphne import errors where importing `daphne.server` didn't work due to a bad linter fix.

Backwards Incompatible Changes

None.

3.6.13 2.1.5 Release Notes

Channels 2.1.5 is another bugfix release in the 2.1 series.

Bugfixes & Small Changes

- Django middleware caching now works on Django 1.11 and Django 2.0. The previous release only ran on 2.1.

Backwards Incompatible Changes

None.

3.6.14 2.1.4 Release Notes

Channels 2.1.4 is another bugfix release in the 2.1 series.

Bugfixes & Small Changes

- Django middleware is now cached rather than instantiated per request resulting in a significant speed improvement. Some middleware took seconds to load and as a result Channels was unusable for HTTP serving before.
- ChannelServerLiveTestCase now serves static files again.
- Improved error message resulting from bad Origin headers.
- `runserver` logging now goes through the Django logging framework to match modern Django.
- Generic consumers can now have non-default channel layers - set the `channel_layer_alias` property on the consumer class
- Improved error when accessing `scope['user']` before it's ready - the user is not accessible in the constructor of ASGI apps as it needs an async environment to load in. Previously it raised a generic error when you tried to access it early; now it tells you more clearly what's happening.

Backwards Incompatible Changes

None.

3.6.15 2.1.3 Release Notes

Channels 2.1.3 is another bugfix release in the 2.1 series.

Bugfixes & Small Changes

- An ALLOWED_ORIGINS value of "*" will now also allow requests without a Host header at all (especially important for tests)
- The request.path value is now correct in cases when a server has SCRIPT_NAME set.
- Errors that happen inside channel listeners inside a runworker or Worker class are now raised rather than suppressed.

Backwards Incompatible Changes

None.

3.6.16 2.1.2 Release Notes

Channels 2.1.2 is another bugfix release in the 2.1 series.

Special thanks to people at the DjangoCon Europe sprints who helped out with several of these fixes.

Major Changes

Session and authentication middleware has been overhauled to be non-blocking. Previously, these middlewares potentially did database or session store access in the synchronous ASGI constructor, meaning they would block the entire event loop while doing so.

Instead, they have now been modified to add LazyObjects into the scope in the places where the session or user will be, and then when the processing goes through their asynchronous portion, those stores are accessed in a non-blocking fashion.

This should be an un-noticeable change for end users, but if you see weird behaviour or an unresolved LazyObject, let us know.

Bugfixes & Small Changes

- AsyncHttpConsumer now has a disconnect() method you can override if you want to perform actions (such as leaving groups) when a long-running HTTP request disconnects.
- URL routing context now includes default arguments from the URLconf in the context's `url_route` key, alongside captured arguments/groups from the URL pattern.
- The FORCE_SCRIPT_NAME setting is now respected in ASGI mode, and lets you override where Django thinks the root URL of your application is mounted.
- ALLOWED_HOSTS is now set correctly during LiveServerTests, meaning you will no longer get `400 Bad Request` errors during these test runs.

Backwards Incompatible Changes

None.

3.6.17 2.1.1 Release Notes

Channels 2.1.1 is a bugfix release for an important bug in the new async authentication code.

Major Changes

None.

Bugfixes & Small Changes

Previously, the object in `scope["user"]` was one of Django's SimpleLazyObjects, which then called our `get_user` async function via `async_to_sync`.

This worked fine when called from SyncConsumers, but because async environments do not run attribute access in an async fashion, when the body of an async consumer tried to call it, the `asgiref` library flagged an error where the code was trying to call a synchronous function during a async context.

To fix this, the User object is now loaded non-lazily on application startup. This introduces a blocking call during the synchronous application constructor, so the ASGI spec has been updated to recommend that constructors for ASGI apps are called in a threadpool and Daphne 2.1.1 implements this and is recommended for use with this release.

Backwards Incompatible Changes

None.

3.6.18 2.1.0 Release Notes

Channels 2.1 brings a few new major changes to Channels as well as some more minor fixes. In addition, if you've not yet seen it, we now have a long-form *tutorial* to better introduce some of the concepts and sync versus async styles of coding.

Major Changes

Async HTTP Consumer

There is a new native-async HTTP consumer class, `channels.generic.http.AsyncHttpConsumer`. This allows much easier writing of long-poll endpoints or other long-lived HTTP connection handling that benefits from native async support.

You can read more about it in the *Consumers* documentation.

WebSocket Consumers

These consumer classes now all have built-in group join and leave functionality, which will make a consumer join all group names that are in the iterable `groups` on the consumer class (this can be a static list or a `@property` method).

In addition, the `accept` methods on both variants now take an optional `subprotocol` argument, which will be sent back to the WebSocket client as the subprotocol the server has selected. The client's advertised subprotocols can, as always, be found in the scope as `scope["subprotocols"]`.

Nested URL Routing

`URLRouter` instances can now be nested inside each other and, like Django's URL handling and `include`, will strip off the matched part of the URL in the outer router and leave only the unmatched portion for the inner router, allowing reusable routing files.

Note that you **cannot** use the Django `include` function inside of the `URLRouter` as it assumes a bit too much about what it is given as its left-hand side and will terminate your regular expression/URL pattern wrongly.

Login and Logout

As well as overhauling the internals of the `AuthMiddleware`, there are now also `login` and `logout` async functions you can call in consumers to log users in and out of the current session.

Due to the way cookies are sent back to clients, these come with some caveats; read more about them and how to use them properly in *Authentication*.

In-Memory Channel Layer

The in-memory channel layer has been extended to have full expiry and group support so it should now be suitable for drop-in replacement for most test scenarios.

Testing

The `ChannelsLiveServerTestCase` has been rewritten to use a new method for launching Daphne that should be more resilient (and faster), and now shares code with the Daphne test suite itself.

Ports are now left up to the operating system to decide rather than being picked from within a set range. It also now supports static files when the Django `staticfiles` app is enabled.

In addition, the Communicator classes have gained a `receive_nothing` method that allows you to assert that the application didn't send anything, rather than writing this yourself using exception handling. See more in the *Testing* documentation.

Origin header validation

As well as removing the `print` statements that accidentally got into the last release, this has been overhauled to more correctly match against headers according to the Origin header spec and align with Django's `ALLOWED_HOSTS` setting.

It can now also enforce protocol (`http` versus `https`) and port, both optionally.

Bugfixes & Small Changes

- `print` statements that accidentally got left in the `Origin` validation code were removed.
- The `runserver` command now shows the version of Channels you are running.
- Orphaned tasks that may have caused warnings during test runs or occasionally live site traffic are now correctly killed off rather than letting them die later on and print warning messages.
- `WebsocketCommunicator` now accepts a query string passed into the constructor and adds it to the scope rather than just ignoring it.
- Test handlers will correctly handle changing the `CHANNEL_LAYERS` setting via decorators and wipe the internal channel layer cache.
- `SessionMiddleware` can be safely nested inside itself rather than causing a runtime error.

Backwards Incompatible Changes

- The format taken by the `OriginValidator` for its domains has changed and `*.example.com` is no longer allowed; instead, use `.example.com` to match a domain and all its subdomains.
- If you previously nested `URLRouter` instances inside each other both would have been matching on the full URL before, whereas now they will match on the unmatched portion of the URL, meaning your URL routes would break if you had intended this usage.

3.6.19 2.0.2 Release Notes

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

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

New Features

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

Bugfixes

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

Backwards Incompatible Changes

None.

3.6.20 2.0.1 Release Notes

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

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

New Features

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

Bugfixes

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

Backwards Incompatible Changes

None.

3.6.21 2.0.0 Release Notes

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

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

More detailed information on the changes and tips on how to port your applications can be found in our `/one-to-two` documentation in the 2.x docs version.

Backwards Incompatible Changes

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

A migration guide is available in the 2.x docs version, and a lot of the basic concepts are the same, but the basic class structure and imports have changed.

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

3.6.22 1.1.6 Release Notes

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

Major Changes

None.

Minor Changes & Bugfixes

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

Backwards Incompatible Changes

None.

3.6.23 1.1.5 Release Notes

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

Major Changes

None.

Minor Changes & Bugfixes

- The Daphne dependency requirement was bumped to 1.3.0.

Backwards Incompatible Changes

None.

3.6.24 1.1.4 Release Notes

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

Major Changes

None.

Minor Changes & Bugfixes

- Pending messages correctly handle retries in backlog situations
- Workers in threading mode now respond to ctrl-C and gracefully exit.
- `request.meta['QUERY_STRING']` is now correctly encoded at all times.
- Test client improvements
- `ChannelServerLiveTestCase` added, allows an equivalent of the Django `LiveTestCase`.
- Decorator added to check `Origin` headers (`allowed_hosts_only`)
- New `TEST_CONFIG` setting in `CHANNEL_LAYERS` that allows varying of the channel layer for tests (e.g. using a different Redis install)

Backwards Incompatible Changes

None.

3.6.25 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.

3.6.26 1.1.2 Release Notes

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

Major Changes

None.

Minor Changes & Bugfixes

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

Backwards Incompatible Changes

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

3.6.27 1.1.1 Release Notes

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

Major Changes

None.

Minor Changes & Bugfixes

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

Backwards Incompatible Changes

None.

3.6.28 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 javascript documentation.
- Test classes have been moved from `channels.tests` to `channels.test` to better match Django. Old imports from `channels.tests` will continue to work but will trigger a deprecation warning, and `channels.tests` will be removed completely in version 1.3.

Minor Changes & Bugfixes

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

Backwards Incompatible Changes

None.

3.6.29 1.0.3 Release Notes

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

Changes

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

Backwards Incompatible Changes

None.

3.6.30 1.0.2 Release Notes

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

Changes

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

Backwards Incompatible Changes

None.

3.6.31 1.0.1 Release Notes

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

Changes

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

Backwards Incompatible Changes

None.

3.6.32 1.0.0 Release Notes

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

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

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

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

Major Features

Channels 1.0 introduces a couple of new major features.

WebSocket accept/reject flow

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

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

This has several advantages:

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

Consumer Atomicity

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

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

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

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

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

Databinding Group/Action Overhaul

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

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

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

Demultiplexer Overhaul

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

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

For more, read the updated `/generic` docs.

Delay Server

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

Minor Changes

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

Backwards Incompatible Changes

Connect Consumers

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

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

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

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

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

Databinding group_names

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

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

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

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

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

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

Demultiplexers

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

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

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

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