WEBSOCKET GUIDE

IDEO-LAB - 2025

WebSocket

Before installing anything, ensure your package lists are up-to-date:

```bash
sudo apt update
```

2. Install Python and Pip (if not already installed)

If you don't have Python and `pip` installed, install them using the following commands:

```bash
sudo apt install python3
sudo apt install python3-pip
```

3. Install the `websockets` library

You can install the `websockets` Python library using `pip`:

```bash
pip3 install websockets
```

4. Verify installation

To check if the WebSocket library is installed correctly, you can open the Python shell and try importing it:

bash Copier le code

```
python3
>>> import websockets
>>> exit()
```

If no errors occur, WebSockets has been installed successfully.

Optional: Install WebSocket in Node.js

If you prefer to work with WebSockets in JavaScript (using Node.js), follow these additional steps:

1. **Install Node.js:**

bash Copier le code

```
sudo apt install nodejs
sudo apt install npm
```

2. **Install the `ws` WebSocket package:**

bash

```
npm install ws
```

Now you're ready to work with WebSockets in Python or Node.js on your Ubuntu

WEBSOCKET & DJANGO

AGENDA

- 1. **Why Use Django Channels for WebSockets?**
- 2. **Installing Django Channels**
- 3. Configuring Django Settings (settings.py)
- 4. Creating the ASGI Configuration (asgi.py)
- 5. Defining WebSocket Routes (routing.py)
- 6. Creating a WebSocket Consumer (consumers.py)
- 7. JavaScript Client for WebSocket
- 8. Scaling with Redis (Optional)
- 9. Running the Application

Summary of Key Components:

- **Django Channels:** Adds WebSocket support to Django.
- **Consumers:** Handle WebSocket events (connect, receive, disconnect).
- **ASGI (Asynchronous Server Gateway Interface):** Allows Django to handle asynchronous communication like WebSockets.
- **Redis (Optional):** Used for scaling real-time communication across multiple instances.

1. **Why Use Django Channels for WebSockets?**

Django by default is synchronous and handles HTTP requests in a request-response cycle. However, WebSockets require a persistent connection to the server for real-time communication. Django Channels extends Django's functionality to support handling WebSockets, HTTP2, and other real-time protocols by using an **asynchronous server gateway interface (ASGI)**.

To integrate WebSockets with Django, you typically use **Django Channels**, which extends Django's capabilities to handle WebSockets and other real-time protocols. Here's a step-by-step guide to setting up WebSockets in Django using Django Channels.

Step-by-Step Guide:

1. **Install Django Channels**

First, install Django Channels:

bash Copier le code

```bash
pip install channels
```

2. **Update Django settings**

Modify the `settings.py` file to include `channels` and configure the ASGI application:

- Add `channels` to the list of installed apps:

python Copier le code

```python
# settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    # other Django apps,
    'channels',  # Add channels here
]
```

3. Configuring Django Settings (`settings.py`)

After installing, you need to modify your `settings.py` to tell Django to use Channels:

- **INSTALLED_APPS**: Add `'channels'` to your list of installed apps so Django knows it will use Channels for handling WebSocket connections.

python — Copier le code

```python
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'channels',  # Add this line
]
```

- **ASGI_APPLICATION**: This is similar to the standard `WSGI_APPLICATION` but specific for ASGI. ASGI is the protocol that allows Django to handle asynchronous requests such as WebSocket connections. Add the following line:

python — Copier le code

```python
ASGI_APPLICATION = 'your_project_name.asgi.application'
```

3. **Create the ASGI configuration file**

In your Django project directory, you should create a file called `asgi.py`. This file will configure the ASGI application to handle WebSocket connections.

python Copier le code

```python
# asgi.py

import os
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from your_app_name import routing  # Import your routing configuration

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

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter(
            routing.websocket_urlpatterns  # Define WebSocket routes here
        )
    ),
})
```

- Define the ASGI application. You'll also need to specify an ASGI application, which will be used for handling WebSocket connections:

python Copier le code

```python
# settings.py

ASGI_APPLICATION = 'your_project_name.asgi.application'  # Replace with your project na
```

4. Creating the ASGI Configuration (`asgi.py`)

You need to configure how Django Channels handles WebSocket connections. Here's a breakdown of the `asgi.py` file:

python Copier le code

```python
import os
from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from your_app_name import routing  # Import the routing module

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

application = ProtocolTypeRouter({
    "http": get_asgi_application(),  # Handles HTTP traffic
    "websocket": AuthMiddlewareStack(  # Handles WebSocket traffic
        URLRouter(
            routing.websocket_urlpatterns  # Refers to WebSocket routes in routing.py
        )
    ),
})
```

- **ProtocolTypeRouter:** This router checks the type of connection (HTTP or WebSocket) and routes it accordingly.
- **AuthMiddlewareStack:** This middleware ensures WebSocket connections go through Django's authentication system, allowing you to handle authenticated users.
- **URLRouter:** This is where WebSocket URL routing is configured (from `routing.py`).

4. **Create a routing configuration**

You'll need a routing configuration to define the WebSocket routes. In your app directory, create a file named `routing.py`:

python Copier le code

```python
# routing.py

from django.urls import path
from your_app_name import consumers  # Import your WebSocket consumers

websocket_urlpatterns = [
    path('ws/some_path/', consumers.YourConsumer.as_asgi()),  # Define WebSocket routes
]
```

5. Defining WebSocket Routes (`routing.py`)

Next, you define the URLs for WebSocket connections. This is done in a new file called `routing.py`, which is similar to Django's `urls.py`, but specific to WebSocket routes.

Example `routing.py`:

python Copier le code

```python
from django.urls import path
from your_app_name import consumers  # Import WebSocket consumer

websocket_urlpatterns = [
    path('ws/some_path/', consumers.YourConsumer.as_asgi()),  # Define WebSocket URL pa
]
```

- **path('ws/some_path/', consumers.YourConsumer.as_asgi())**: This maps the WebSocket route (`ws/some_path/`) to the consumer (`YourConsumer`), which will handle the WebSocket connection.

5. **Create a WebSocket consumer**

In the same app, create a consumer class that will handle WebSocket connections. For example, create a file `consumers.py`:

python Copier le code

```python
# consumers.py

import json
from channels.generic.websocket import AsyncWebsocketConsumer

class YourConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        await self.accept()

    async def disconnect(self, close_code):
        pass

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

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

This simple consumer accepts WebSocket connections, echoes back any messages it receives, and closes the connection on disconnect.

In this consumer:

- **connect()**: Called when a WebSocket connection is initiated. `await self.accept()` allows the connection to proceed.
- **disconnect()**: Called when the WebSocket connection is closed.
- **receive()**: Called whenever a message is received from the WebSocket client. The message is then echoed back to the client using `self.send()`.

This simple consumer accepts WebSocket connections, echoes back any messages it receives, and closes the connection on disconnect.

6. **Configure Redis (optional but recommended for scaling)**

For production or larger projects, you'll want to use a message broker like **Redis** to manage connections efficiently:

1. Install Redis and `channels_redis`:

```bash
sudo apt install redis-server
pip install channels_redis
```

2. Update your `settings.py` to configure Channels to use Redis as the message broker:

```python
# settings.py

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

7. **Run your Django project**

Start your Django project with the following command to run the ASGI server:

bash Copier le code

```
python manage.py runserver
```

This will start your Django project, with WebSocket support via Django Channels.

Example of WebSocket connection in JavaScript

Here's how you can connect to the WebSocket in your frontend:

javascript Copier le code

```
var socket = new WebSocket('ws://localhost:8000/ws/some_path/');

socket.onmessage = function(e) {
    const data = JSON.parse(e.data);
    console.log(data.message);
};

socket.onopen = function(e) {
    socket.send(JSON.stringify({
        'message': 'Hello WebSocket!'
    }));
};

socket.onclose = function(e) {
    console.log('WebSocket connection closed');
};
```

7. JavaScript Client for WebSocket

To connect to the WebSocket from the frontend, you can use JavaScript's built-in WebSocket API.

Here's a sample JavaScript code:

javascript Copier le code

```javascript
var socket = new WebSocket('ws://localhost:8000/ws/some_path/');

socket.onmessage = function(event) {
    var data = JSON.parse(event.data);
    console.log(data.message);  // Log the message received from the server
};

socket.onopen = function(event) {
    socket.send(JSON.stringify({
        'message': 'Hello WebSocket!'
    }));
};

socket.onclose = function(event) {
    console.log('WebSocket connection closed');
};
```

In this example:

- A WebSocket connection is established using the URL `ws://localhost:8000/ws/some_path/`.
- When the WebSocket is open (`onopen`), a message is sent to the server.
- When a message is received from the server (`onmessage`), it's logged to the console.
- The connection is gracefully closed with `onclose`.

8. Scaling with Redis (Optional)

When you scale WebSockets to production, it's best to use Redis as a message broker. Django Channels supports Redis for handling channel layers (the communication between multiple Django instances for WebSockets).

1. **Install Redis:**

bash — Copier le code

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

2. **Install Redis support for Django Channels:**

bash — Copier le code

```
pip install channels_redis
```

3. **Update** `settings.py`:

python — Copier le code

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