**CHAT SUMMARY**

**WebSocket** is a communication protocol that provides a full-duplex communication channel over a single, long-lived connection between a client and a server. It is commonly used in scenarios requiring real-time updates, such as chat applications, gaming, financial dashboards, and live feeds

**Key Features:**

1. **Full-Duplex Communication**: Allows simultaneous two-way communication between client and server.
2. **Persistent Connection**: The connection remains open, reducing overhead compared to traditional HTTP requests.
3. **Low Latency**: Ideal for real-time data exchange since it eliminates the need for constant polling.
4. **Lightweight**: Once established, the protocol has minimal overhead compared to other methods like HTTP polling

**WebRTC** is a technology that enables real-time peer-to-peer communication directly between browsers and devices. It facilitates audio, video, and data transfer without requiring an intermediary server.

WebRTC enables direct peer-to-peer communication between browsers. Unlike WebSockets, which require a server to relay messages between clients, WebRTC allows connected devices to communicate directly with each other without an intermediary server.

**Key Features:**

1. **Peer-to-Peer Communication**: Allows direct communication between users without routing through a central server, reducing latency and server load.
2. **Media Streams**: Supports real-time audio and video streaming.
3. **Data Channels**: Enables arbitrary data transfer, such as files, over a secure connection.
4. **Secure**: Communication is encrypted using Secure Real-Time Transport Protocol (SRTP).

ASGI and WSGI are both interfaces for Python web applications, but they are designed to handle different kinds of workloads and have distinct characteristics. Here's a comparison:

**WSGI (Web Server Gateway Interface)**

**WSGI** is a standard interface between web servers and Python web applications or frameworks. It is synchronous, which means it can only handle one request at a time per worker.

**Key Features:**

1. **Synchronous**: Designed for traditional, blocking web applications.
2. **HTTP-only**: Primarily supports HTTP 1.x requests.
3. **Single-threaded or Multi-threaded**: Each worker can handle one request at a time.
4. **Simple**: Easy to understand and implement, making it a long-standing choice for Python web development.

**Use Cases:**

* Ideal for traditional web applications with synchronous workflows, such as **Django** or **Flask** without real-time requirements.

**Limitations:**

* Cannot handle WebSockets or real-time asynchronous communication.
* Not optimized for high-concurrency workloads.

**Popular Servers:**

* **Gunicorn**
* **uWSGI**
* **mod\_wsgi** (for Apache)

**ASGI (Asynchronous Server Gateway Interface)**

**ASGI** is a newer standard that extends WSGI to support asynchronous applications. It is designed for modern web applications requiring real-time features like WebSockets, HTTP/2, and long-lived connections.

**Key Features:**

1. **Asynchronous**: Supports both synchronous and asynchronous programming models.
2. **WebSockets and More**: Handles HTTP, WebSockets, and other protocols.
3. **High Concurrency**: Can handle thousands of connections concurrently due to its non-blocking architecture.
4. **Scalable**: Suitable for modern web use cases involving real-time communication.

**Use Cases:**

* Real-time applications like **chat apps**, **live notifications**, or **IoT dashboards**.
* Applications using frameworks like **Django (with async capabilities)**, **FastAPI**, or **Starlette**.

**Benefits:**

* Flexibility to handle both synchronous and asynchronous workloads.
* Future-proof for modern web standards.

**Popular Servers:**

* **Uvicorn**
* **Daphne**
* **Hypercorn**

**Key Differences**

| **Feature** | **WSGI** | **ASGI** |
| --- | --- | --- |
| **Design** | Synchronous | Asynchronous (also supports sync) |
| **Protocols Supported** | HTTP 1.x only | HTTP 1.x, HTTP/2, WebSockets |
| **Concurrency** | Limited to one request per worker | Handles multiple requests per worker |
| **Real-time Support** | No | Yes |
| **Use Case** | Traditional web apps | Real-time and modern apps |
| **Example Frameworks** | Django, Flask | Django (async), FastAPI, Starlette |
| **Performance** | May block under heavy loads | Better concurrency and scalability |

**When to Use Which?**

* **Use WSGI**:
  + If your application is entirely synchronous and doesn’t require real-time communication.
  + If you are using traditional frameworks like Django (pre-async support) or Flask.
* **Use ASGI**:
  + If your application has real-time requirements, such as WebSocket support or needs to handle thousands of connections concurrently.
  + If you're building modern applications with FastAPI, Starlette, or Django with async capabilities.

**Transition Note:**

Django has adopted ASGI since version 3.0, making it compatible with both WSGI and ASGI, allowing developers to choose based on their needs.

**Django Channels** is an extension of Django that enables handling **real-time protocols** like **WebSockets**, **HTTP/2**, and **background tasks**. It uses **ASGI** for asynchronous communication, making Django capable of supporting live chat, notifications, and IoT systems.

**Key Features:**

1. **WebSocket Support**: Enables real-time communication.
2. **Background Tasks**: Handles long-running jobs outside the main request cycle.
3. **Scalability**: Uses Redis as a **message queue** for handling multiple processes.
4. **Consumers**: Async equivalents of views for WebSocket or other protocols.

**Example Use Cases:**

* Live chat apps.
* Real-time dashboards.
* Notifications for users.

**How It Works:**

* Replaces WSGI with **ASGI** for async support.
* Uses **consumers** to handle WebSocket events like connect, receive, and disconnect.
* **Redis channel layer** enables communication between app instances.

**Why Use It?**

* Adds real-time capabilities to Django.
* Scales well with Redis.
* Ideal for modern apps needing real-time updates.

**Key Difference:**

* **WebSocket**: Protocol for real-time communication.
* **ASGI**: Standard interface enabling async protocols (including WebSockets).
* **Django Channels**: A Django-specific tool built on ASGI to simplify WebSocket and async handling.

To upgrade your Django app to accept WebSocket protocols, the process is similar to handling HTTP requests but tailored for WebSocket communication. Here's a step-by-step breakdown:

**1. Define a Consumer**

A **consumer** in Django Channels is similar to a Django view but is designed to handle WebSocket events like connect, receive, and disconnect.

**Example:**

Create a consumer for WebSocket handling:

# myapp/consumers.py

from channels.generic.websocket import AsyncWebsocketConsumer

import json

class ChatConsumer(AsyncWebsocketConsumer):

async def connect(self):

# Accept the WebSocket connection

await self.accept()

print("WebSocket connected")

async def disconnect(self, close\_code):

# Handle WebSocket disconnection

print("WebSocket disconnected")

async def receive(self, text\_data):

# Handle messages received from the client

data = json.loads(text\_data)

message = data.get('message', 'No message sent')

# Send a message back to the client

await self.send(json.dumps({'message': message}))

**2. Configure WebSocket URLs (Routes)**

Instead of urls.py, WebSocket routes are defined in a separate routing file, typically called routing.py. Here, you link WebSocket paths to their respective consumers.

**Example:**

Define WebSocket routes for your app:

# myapp/routing.py

from django.urls import path

from . import consumers

websocket\_urlpatterns = [

path('ws/chat/', consumers.ChatConsumer.as\_asgi()), # WebSocket endpoint

]

**3. Add App Routes to the Project's ASGI Application**

Update your project's asgi.py to include the app's WebSocket routing. This is similar to adding app URLs in Django’s urls.py.

**Example:**

Modify asgi.py to include the WebSocket routes:

# myproject/asgi.py

import os

from django.core.asgi import get\_asgi\_application

from channels.routing import ProtocolTypeRouter, URLRouter

from channels.auth import AuthMiddlewareStack

from myapp.routing import websocket\_urlpatterns

os.environ.setdefault('DJANGO\_SETTINGS\_MODULE', 'myproject.settings')

application = ProtocolTypeRouter({

"http": get\_asgi\_application(), # Handles HTTP requests

"websocket": AuthMiddlewareStack( # Handles WebSocket connections

URLRouter(

websocket\_urlpatterns # Include WebSocket routes

)

),

})

**Summary of Steps**

1. **Define a Consumer**: Create a class-based consumer to handle WebSocket events.
2. **Configure WebSocket URLs**: Map WebSocket endpoints to consumers in routing.py.
3. **Add Routes to ASGI**: Include the WebSocket routes in the project's asgi.py.

**How It Works**

1. When a client connects to a WebSocket URL (e.g., ws://example.com/ws/chat/), the ChatConsumer handles the connection.
2. Incoming messages are processed by the receive method.
3. Messages can be sent back to the client using the send method.
4. The connection is managed by Django Channels' ASGI application.

This setup enables your Django app to handle WebSocket protocols alongside traditional HTTP requests!

WebSockets and WebRTC are both technologies used for real-time communication over the web, but they serve different purposes and have distinct characteristics. Here's a comparison:

| **Feature** | **WebSockets** | **WebRTC** |
| --- | --- | --- |
| **Purpose** | General-purpose, full-duplex communication channel | Real-time peer-to-peer communication for audio, video, and data |
| **Communication Type** | Client-server communication | Peer-to-peer communication |
| **Use Case** | Chat applications, notifications, real-time updates | Video conferencing, file sharing, gaming, P2P data |
| **Protocol** | Uses WebSocket protocol (ws:// or wss://) over TCP | Uses several protocols, including SRTP, SCTP, and ICE |
| **Transport Layer** | TCP | Primarily UDP (though fallback to TCP is possible) |
| **Latency** | Higher compared to WebRTC due to TCP overhead | Lower latency because of UDP's efficiency |
| **Data Types** | Arbitrary data in text or binary format | Optimized for media streams and supports arbitrary data |
| **Server Dependency** | Requires a server to relay messages | Establishes direct connections after signaling |
| **Security** | Encrypted using TLS | Encrypted using DTLS and SRTP |
| **Ease of Use** | Easier to set up with client-server interaction | More complex setup involving signaling servers |

**Key Takeaways:**

* **WebSockets** are ideal for real-time applications where a persistent connection between client and server is needed (e.g., chat systems or live updates).
* **WebRTC** is tailored for peer-to-peer communication, especially for media streaming, with built-in capabilities for low-latency video, audio, and data transfer.

So in short:

1. We will use HTTP requests for displaying regular html pages

2. We will use WebSockets for sending text messages

3. We will use WebRTC for video call

In ASGI (Asynchronous Server Gateway Interface), **channels** play a crucial role in enabling asynchronous communication between different components of your web application. Channels are a fundamental part of the Django Channels framework, which extends Django to handle asynchronous protocols like WebSockets, HTTP2, and more.

**What is a Channel?**

A channel is a queue that allows different parts of an ASGI application to send and receive messages asynchronously. It enables decoupled communication, where one part of the application can produce messages, and another part can consume and process them.

**Redis** (Remote Dictionary Server) is an open-source, in-memory data structure store that is primarily used as a database, cache, and message broker. It is known for its high performance, low latency, and flexibility, making it a popular choice for various real-time applications.

how to check version of channel

pip show channels