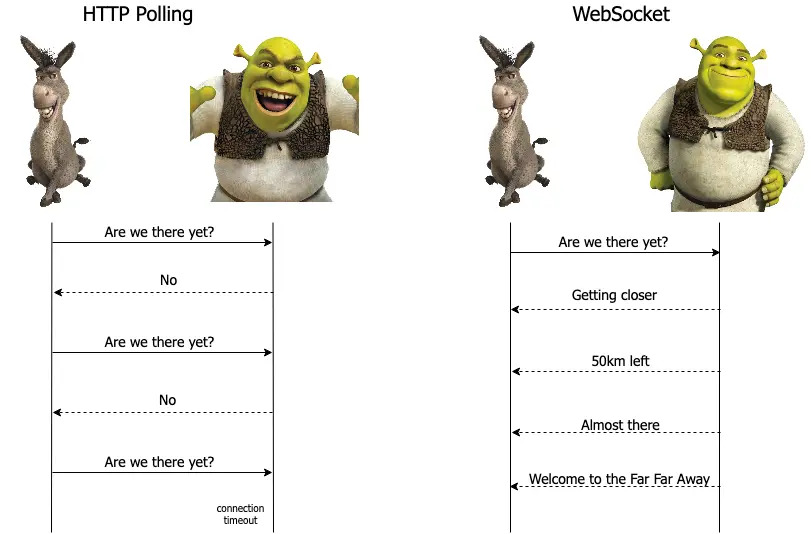


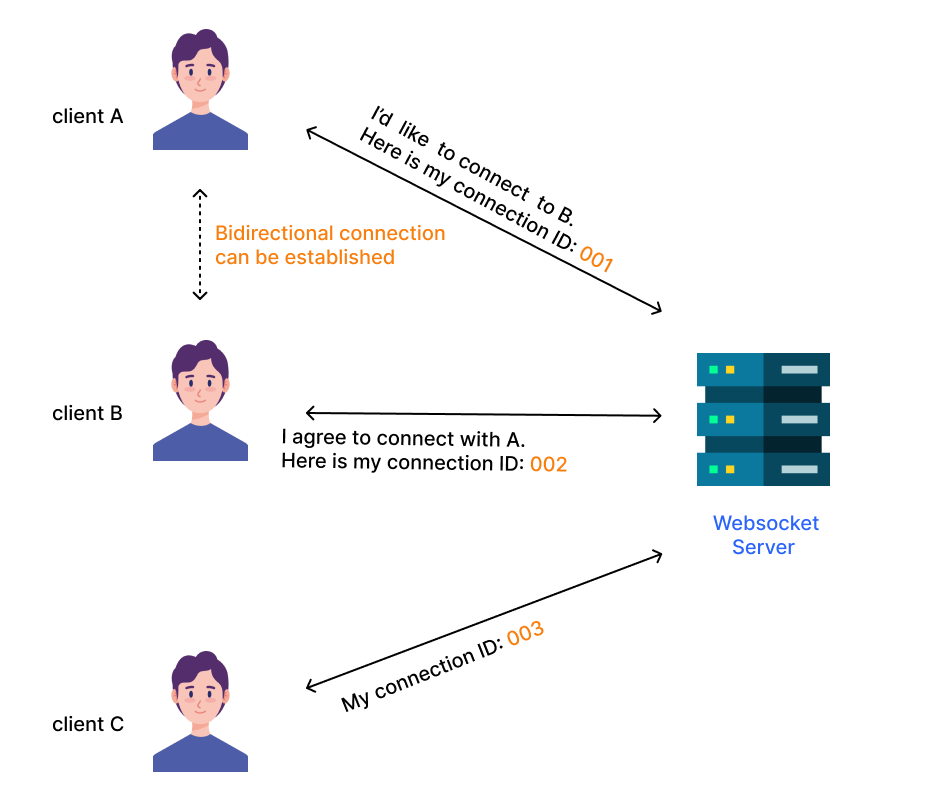
WebSocket provide a persistent connection between a client (such as a web browser) and a server, allowing for **bi-directional communication**. WebSocket maintain a **constant connection**, enabling both the client and server to send data to each other at any time without the overhead of repeated handshakes.

WebSocket are used in various scenarios where real-time, interactive communication is required, such as chat applications, online gaming, stock market tracking, collaborative editing tools, and more.

They offer several advantages over traditional HTTP requests, including **reduced latency**, lower overhead, and improved efficiency for applications that require **frequent data exchange** between the client and server.



**Multiple connections and private conversions**

****

WebSocket server facilitating bidirectional connections between clients

**1. Client A Initiates Connection:**

* Client A wants to establish a connection with Client B.
* Client A sends a message to the WebSocket server indicating the desire to connect to Client B. This message includes Client A's connection ID (e.g., 001).

**2. Server Receives and Forwards Request:**

* The WebSocket server receives Client A's request.
* The server then relays this connection request to Client B, informing them that Client A wants to connect.

**3. Client B Accepts Connection:**

* Client B receives the connection request from the server.
* If Client B agrees to connect with Client A, they send a response back to the WebSocket server. This response includes Client B's connection ID (e.g., 002).

**4. Server Establishes Bidirectional Connection:**

* Upon receiving Client B's acceptance, the WebSocket server acknowledges that both clients agree to connect.
* A bidirectional connection is established between Client A and Client B, allowing direct communication between them.

**WebSocket Server:** Acts as a mediator for connection requests and responses, ensuring both clients agree before establishing a direct link.

**Bidirectional Communication:** Once both clients agree, the server facilitates a two-way communication channel, enabling real-time data exchange.

**Connection IDs:** Unique identifiers for each client used to manage connection requests and responses efficiently.

**Python Library: Flask-SocketIO**

**Flask-SocketIO**

**Flask-SocketIO** gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the [SocketIO](http://socket.io/) client libraries in Javascript.

**Concurrency Frameworks**

**asyncio** Asynchronous I/O

* asyncio is a library to write **concurrent** code using the **async/await** syntax.
* asyncio is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc.
* asyncio is often a perfect fit for IO-bound and high-level **structured** network code.
* an event loop runs and is in charge of switching between various coroutines.

**Eventlet**

* Eventlet is a concurrent networking library for Python.
* It uses for highly scalable non-blocking I/O.
* This is very much simplified, but the main point here is that switching out the currently running coroutine is performed **implicitly.**

This Flask application sets up a WebSocket server using Flask-SocketIO. It allows clients to connect, exchange unique client IDs, and send messages to specific clients using these IDs.

**Imports and App Initialization**

**Step 1:**

from flask import Flask, render\_template, request

from flask\_socketio import SocketIO, emit

import uuid

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'secret!'

**Flask:** A lightweight WSGI web application framework.

**Flask-SocketIO:** Extends Flask with WebSocket capabilities.

**uuid:** Used to generate unique client IDs.

**app:** The Flask application instance.

**socketio:** The SocketIO server instance for handling real-time communication.

**app config:** Secret key for session management.

**Step: 2**

socketio = SocketIO(app, *async\_mode*='eventlet', *cors\_allowed\_origins*="\*")

**SocketIO(app)**

* **app:** The Flask application instance. This binds the SocketIO server to your Flask app, allowing it to handle WebSocket events.

**async\_mode='eventlet'**

* **async\_mode:** Specifies the asynchronous mode to be used by Flask-SocketIO.
* **'eventlet':** Eventlet is a concurrent networking library for Python that allows for non-blocking IO and efficient handling of a large number of simultaneous connections.
* Using **'eventlet'** makes your application suitable for handling real-time communication where low latency is important.

**cors\_allowed\_origins="\*"**

* **cors\_allowed\_origins:** Configures Cross-Origin Resource Sharing (CORS) settings.
* **"\*"**: Allows requests from any origin. This is useful during development or if your application needs to accept connections from any domain.

**Step 3:**

The **clients** dictionary is used to map unique client IDs to their corresponding SocketIO session IDs. This mapping is crucial for managing and identifying connected clients, enabling targeted communication between clients.

clients = {}

* **Key:** The unique **client\_id** generated for each connected client.
* **Value:** The SocketIO session ID (**request.sid**) for the client.

**Routes and Event Handlers**

**Step: 4 - Index Route**

@app.route('/')

def index():

    return render\_template('index.html')

Serves the main HTML page to the client when they access the root URL (/).

**Step: 5 - Connect Event**

@socketio.on('connect')

def handle\_connect():

    client\_id = str(uuid.uuid4())

    clients[client\_id] = request.sid

    emit('client\_id', {'clientId': client\_id})

    print(f"Client connected: {client\_id}")

* **connect:** Triggered when a client establishes a connection.
* Generates a unique **client\_id** using **uuid**.
* Stores the mapping of **client\_id** to the client's SocketIO session ID (**request.sid**) in the clients dictionary.
* Sends the generated **client\_id** back to the client.
* Logs the connection event.

**Step: 6 - Disconnect Event**

@socketio.on('disconnect')

def handle\_disconnect():

    client\_id = None

    for cid, sid in clients.items():

        if sid == request.sid:

            client\_id = cid

            break

    if client\_id:

        del clients[client\_id]

        print(f"Client disconnected: {client\_id}")

* **disconnect:** Triggered when a client disconnects.
* Finds the **client\_id** corresponding to the client's session ID.
* Removes the client from the **clients** dictionary.
* Logs the disconnection event.

**Step: 7 – Message Event**

@socketio.on('message')

def handle\_message(*data*):

    target\_client\_id = *data*.get('targetClientId')

    message\_content = *data*.get('message')

    if target\_client\_id in clients:

        target\_sid = clients[target\_client\_id]

        emit('message', {'from': *data*.get('from'), 'content': message\_content}, *room*=target\_sid)

    else:

        emit('message', {'error': 'Target client ID not found'}, *room*=request.sid)

The **handle\_message** function manages the event when a client sends a message intended for another client. It ensures that the message is delivered to the correct recipient based on the target client ID.

**Code Breakdown**

@socketio.on('message')

def handle\_message(*data*):

    target\_client\_id = *data*.get('targetClientId')

    message\_content = *data*.get('message')

* **@socketio.on('message'):** This decorator registers the function as an event handler for the message event. This event is triggered when a client sends a message.
* **def handle\_message(data)::** Defines the function to handle the message event. The data parameter contains the message data sent by the client.
* **target\_client\_id = data.get('targetClientId'):** Extracts the target client ID from the message data. This ID specifies the intended recipient of the message.
* **message\_content = data.get('message'):** Extracts the actual message content from the message data.

**Checking if the Target Client Exists**

if target\_client\_id in clients:

* **if target\_client\_id in clients::** Checks if the target client ID exists in the clients dictionary. This ensures that the target client is currently connected.

**Sending the Message to the Target Client**

target\_sid = clients[target\_client\_id]

        emit('message', {'from': *data*.get('from'), 'content': message\_content}, *room*=target\_sid)

* **target\_sid = clients[target\_client\_id]:** Retrieves the SocketIO session ID (target\_sid) of the target client from the clients dictionary.
* Sends the message to the target client. The **emit** function sends the message with the event name 'message' and includes the sender's ID (**data.get('from')**) and the message content **(message\_content**). The **room=target\_sid** parameter ensures that the message is sent to the specific session ID of the target client.

**Handling the Case Where the Target Client is Not Found**

else:

        emit('message', {'error': 'Target client ID not found'}, *room*=request.sid)

* **else::** If the target client ID is not found in the clients dictionary.
* Sends an error message back to the sender (the client who sent the original message). The **emit** function sends a message with the event name 'message' and includes an error message. The **room=request.sid** parameter ensures that the error message is sent back to the sender's session ID.

**Step: 8 - Running the Application**

if \_\_name\_\_ == '\_\_main\_\_':

    socketio.run(app, *debug*=True)

Runs the Flask application with SocketIO support in debug mode.

**Summary**

* Assigning unique IDs to connected clients.
* Maintaining a dictionary to track connected clients and their session IDs.
* Enabling clients to send messages to specific clients using these unique IDs.
* Handling client connection and disconnection events to manage the clients dictionary dynamically.