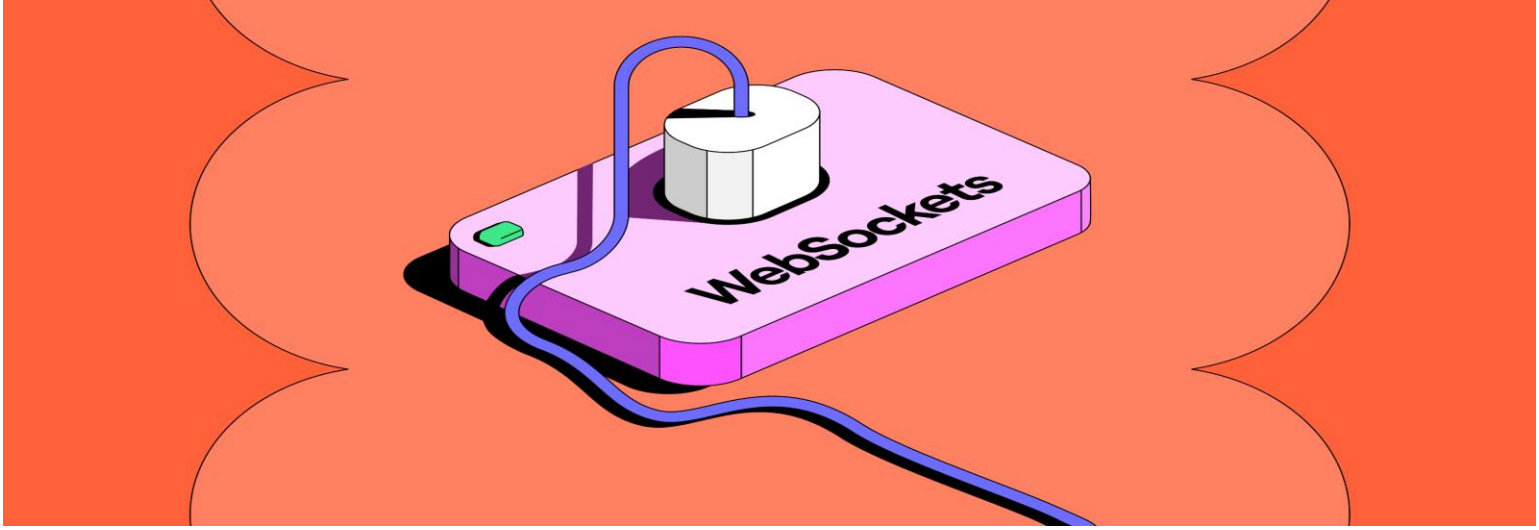
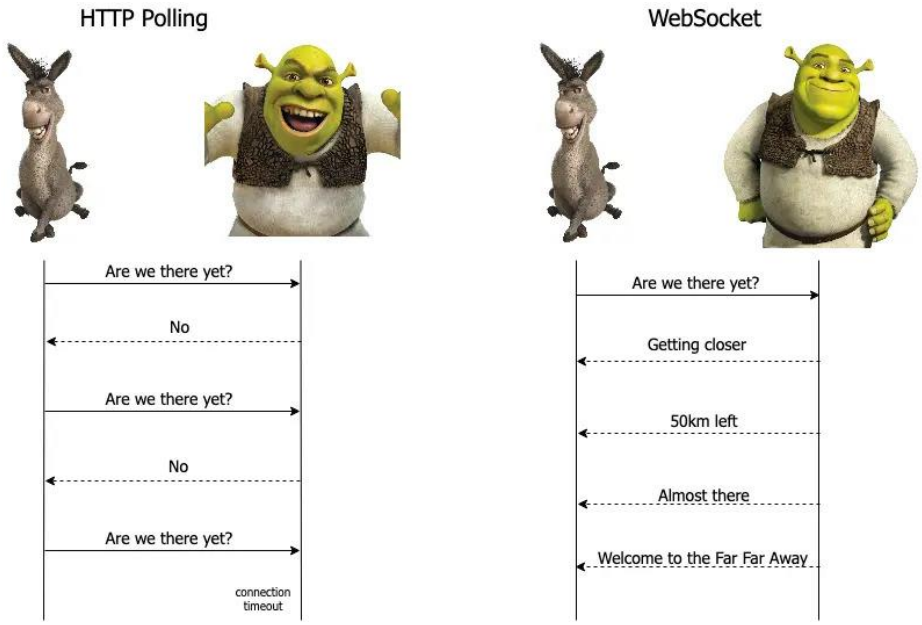


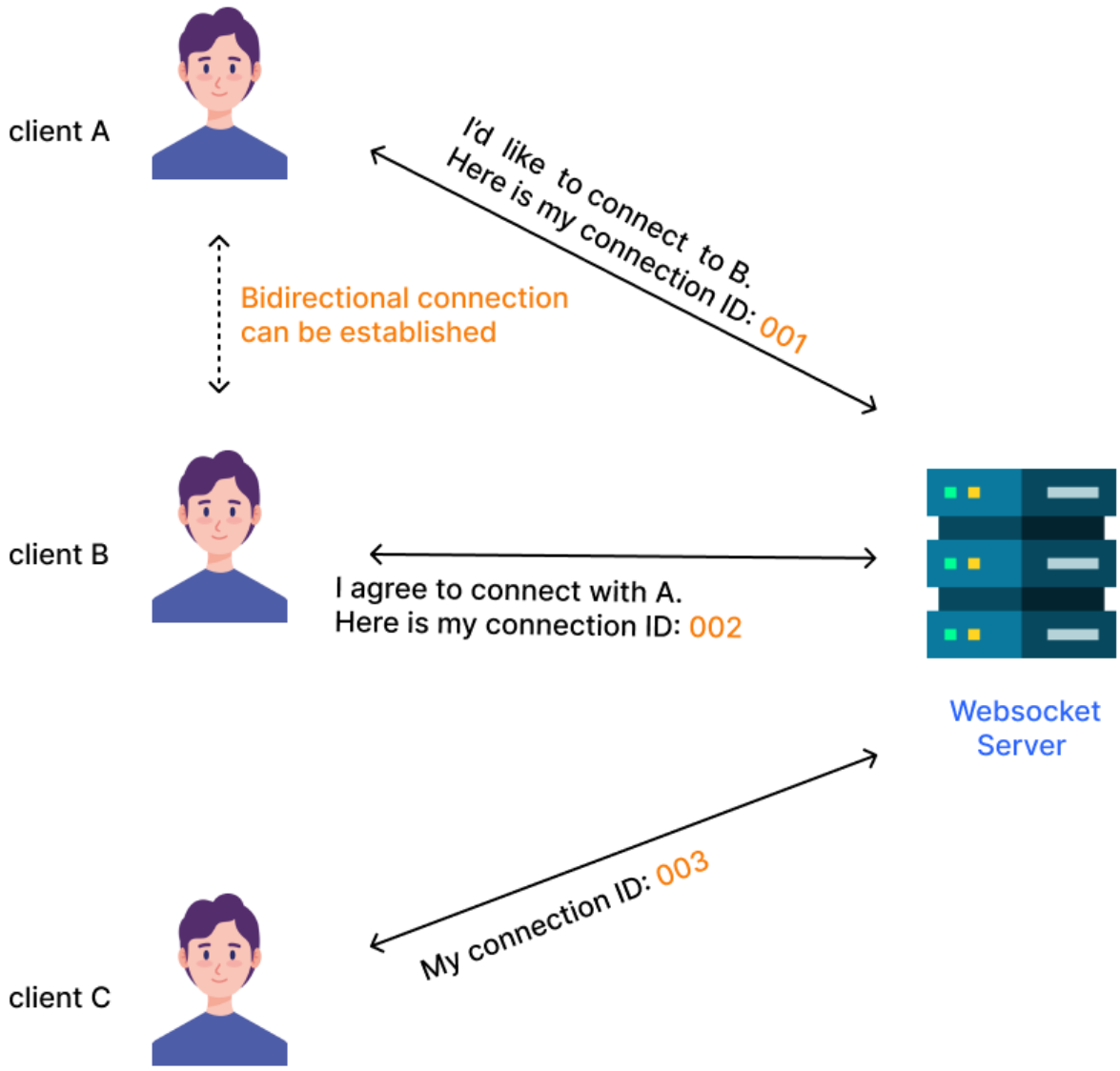
WebSocket provide a persistent connection between a client (such as a web browser) and a server, allowing for **bi-directional communication**. WebSocket maintain a **constant connection**, enabling both the client and server to send data to each other at any time without the overhead of repeated handshakes.

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 Architecture



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 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.

UI Design

This code sets up a basic client-side WebSocket communication using Socket.io library in a web application. Let's break down the code:

HTML Structure

Templates/index.html

```
<h1>WebSocket Client</h1>
  <div id="status">Connecting...</div>
  <div id="client-id">Client ID: N/A</div>
  <div id="messages">
    <h2>Messages</h2>
    <ul id="messages-list"></ul>
  </div>
  <div id="send-message">
    <h2>Send Message</h2>
    <form id="message-form">
      <label for="target-client-id">Target Client ID:</label>
      <input type="text" id="target-client-id" required>
      <label for="message-content">Message:</label>
      <input type="text" id="message-content" required>
      <button type="submit">Send</button>
    </form>
  </div>
```

Code Breakdown

Main Heading:

```
<h1>WebSocket Client</h1>
```

Displays the title of the web page.

Connection Status:

```
<div id="status">Connecting...</div>
```

- Initially shows "Connecting..." indicating that the client is attempting to establish a connection with the WebSocket server.
- The status will be updated dynamically based on the connection state (connected, disconnected, etc.).

Client ID Display:

```
<div id="client-id">Client ID: N/A</div>
```

Displays the client's unique ID once it is assigned by the server. Initially set to "N/A".

Messages Section:

```
<div id="messages">
    <h2>Messages</h2>
    <ul id="messages-list"></ul>
  </div>
```

- Contains a subheading "Messages".
- An unordered list (<**ul**>) with the ID **messages-list** will hold the list of messages exchanged. Each message will be added as a list item (<**li**>).

Send Message Section:

```
<div id="send-message">
    <h2>Send Message</h2>
    <form id="message-form">
      <label for="target-client-id">Target Client ID:</label>
      <input type="text" id="target-client-id" required>
      <label for="message-content">Message:</label>
      <input type="text" id="message-content" required>
      <button type="submit">Send</button>
    </form>
  </div>
```

- Contains a subheading "Send Message".
- A form (<**form**>) with the ID **message-form** for sending messages.
- Label and input field for the target client ID.
- Label and input field for the message content.
- Submit button to send the message.

SocketIO Client Library

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"></scri
pt>
```

Socket.IO client library from a content delivery network (CDN).

JavaScript Structure

This JavaScript code sets up a WebSocket client using Socket.IO and handles various events and interactions. Let's break it down:

Initialization:

```
const socket = io();
```

Creates a WebSocket connection using Socket.IO.

DOM Element References:

```
const statusDiv = document.getElementById('status');
const clientIdDiv = document.getElementById('client-id');
const messagesList = document.getElementById('messages-list');
const messageForm = document.getElementById('message-form');
const targetClientIdInput = document.getElementById('target-client-id');
const messageContentInput = document.getElementById('message-content');
```

Retrieves references to various HTML elements by their IDs, which are essential for interacting with the DOM.

Event Listeners:

The code sets up event listeners for different WebSocket events and for the form submission.

Connect Event

```
socket.on('connect', () => {
      statusDiv.textContent = 'Connected to the WebSocket server';
});
```

connect: Triggered when the WebSocket connection is established. It updates the status display to show that the client is connected.

Client Id Event

```
socket.on('client_id', (data) => {
      clientId = data.clientId;
      clientIdDiv.textContent = `Client ID: ${clientId}`;
    });
```

client_id: Triggered when the server assigns a client ID to the connected client. It updates the client ID display.

Message Event

```
socket.on('message', (message) => {
      if (message.error) {
        alert(`Error: ${message.error}`);
      } else {
        const listItem = document.createElement('li');
        listItem.classList.add('message', 'received');
        listItem.innerHTML = `<strong>From ${message.from}</strong>:
${message.content}`;
        messagesList.appendChild(listItem);
      }
    });
```

message: Triggered when the client receives a message from the server. It processes the message and updates the UI accordingly.

Disconnect Event

```
socket.on('disconnect', () => {
      statusDiv.textContent = 'Disconnected from the WebSocket server';
    });
```

disconnect: Triggered when the WebSocket connection is closed. It updates the status display to show that the client is disconnected.

Connect Error Event

```
socket.on('connect_error', (error) => {
      console.error('WebSocket error:', error);
      statusDiv.textContent = 'Error occurred. Check console for details.';
    });
```

connect_error: Triggered when there's an error connecting to the WebSocket server. It logs the error and updates the status display to indicate an error occurred.

Form Submission Handling

When the user submits the message form, the code prevents the default form submission behavior, constructs a message object containing the target client ID, sender's ID, and message content, and sends it to the server using the WebSocket connection. It also updates the UI to display the sent message.

```
messageForm.addEventListener('submit', (event) => {
      event.preventDefault();
      const targetClientId = targetClientIdInput.value;
      const messageContent = messageContentInput.value;
      const message = {
        targetClientId: targetClientId,
        from: clientId,
        message: messageContent
      };

      const listItem = document.createElement('li');
      listItem.classList.add('message', 'sent');
      listItem.innerHTML = `<strong>To ${targetClientId}</strong>:
${messageContent}`;
      messagesList.appendChild(listItem);

      socket.send(message);
    });
```

Screenshots:

Client interface