WebSocket Implementation in FastAPI
Advanced Computer Programming

Real-Time Two-Way Communication

DR. MOHAMMED AL-HUBAISHI

https://youtu.be/lQ7s_vJ7naM

Handling multiple WebSocket connections and broadcasting messages

As we said in the introduction to this chapter, a typical use case for WebSockets is to implement real-time communication across multiple clients, such as a chat application. In this configuration, several clients have an open WebSocket tunnel with the server. Thus, the role of the server is to *manage all the client connections and broadcast messages to all of them*: when a user sends a message, the server has to send it to all other clients in their WebSockets. We show you a schema of this principle here:

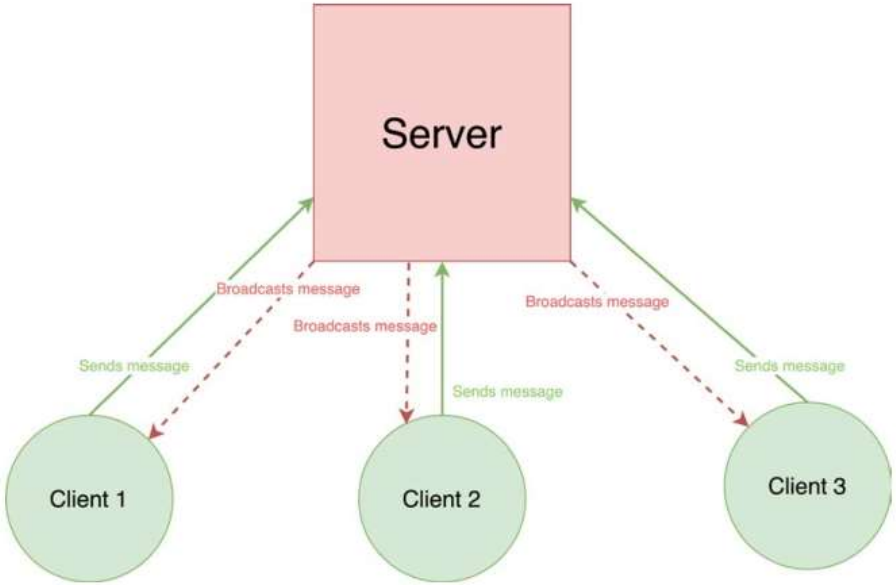


Figure 8.4 – Multiple clients connected through a WebSocket to a server

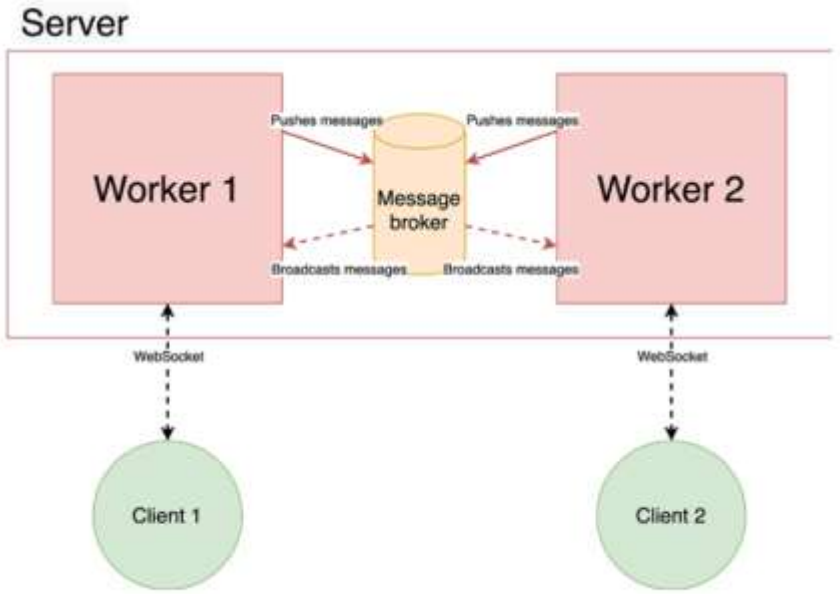


Figure 8.6 – Multiple server workers with a message broker

FastAPI 0.1.0 OAS 3.1

default

GET / Get

Parameters

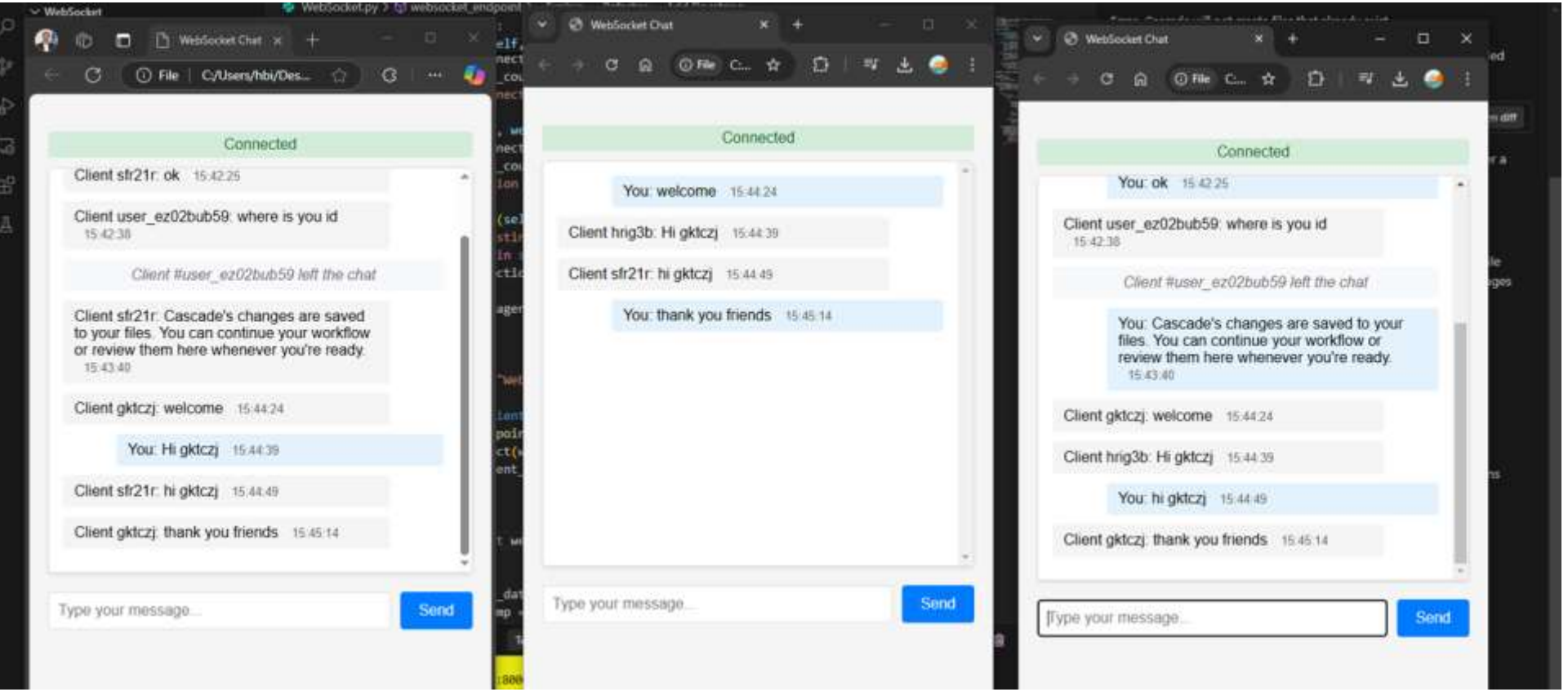
No parameters

Execute

Server response

Code	Details
200	Response body

```
{
  "message": "WebSocket Server is running"
}
```



Introduction to WebSockets in FastAPI

- WebSockets:

- Protocol for full-duplex communication
- Persistent connection between client and server
- Real-time data exchange

- FastAPI Integration:

- Async support for WebSocket connections
- Easy-to-implement WebSocket endpoints
- Efficient handling of multiple connections

WebSocket Protocol

WebSocket Protocol

- Enables two-way communication between client and server
- Maintains persistent connection
- Real-time data transfer
- Built-in support in FastAPI framework
- Ideal for chat applications, live updates, and real-time features

Core WebSocket Functions

- websocket.accept()
  - Accepts incoming WebSocket connection
- websocket.receive_text()
  - Receives text messages from client
- websocket.send_text()
  - Sends text messages to client
- websocket.receive_json()
  - Receives and parses JSON messages
- websocket.send_json()
  - Sends JSON formatted messages

Connection Management Functions

Connection Setup:

- Initialize ConnectionManager
- Handle active connections list
- Track connection count
- Accept new connections

Connection Handling:

- Connect new clients
- Disconnect clients
- Broadcast messages
- Monitor connection state

Message Handling Functions

- receive_text() - Get raw text messages
- receive_json() - Get and parse JSON messages
- send_text() - Send text responses
- send_json() - Send JSON formatted responses
- broadcast() - Send to all connected clients
- JSON parsing and validation
- Error handling for malformed messages

Event Handling Functions

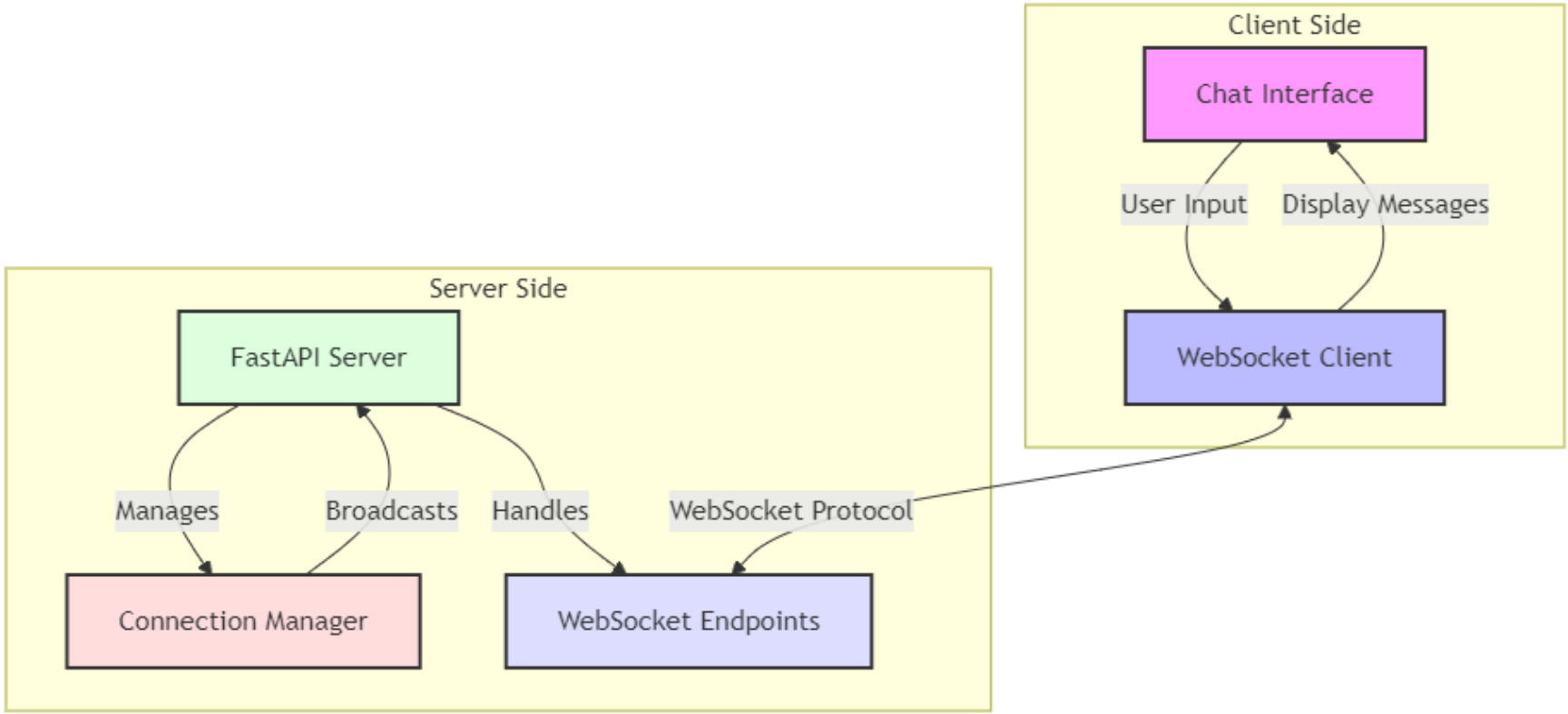
Server-Side Events

- Connection events
- Message reception
- Disconnection handling
- Error management

Client-Side Events

- ws.onopen
- ws.onmessage
- ws.onclose
- ws.onerror

Server and Client sides



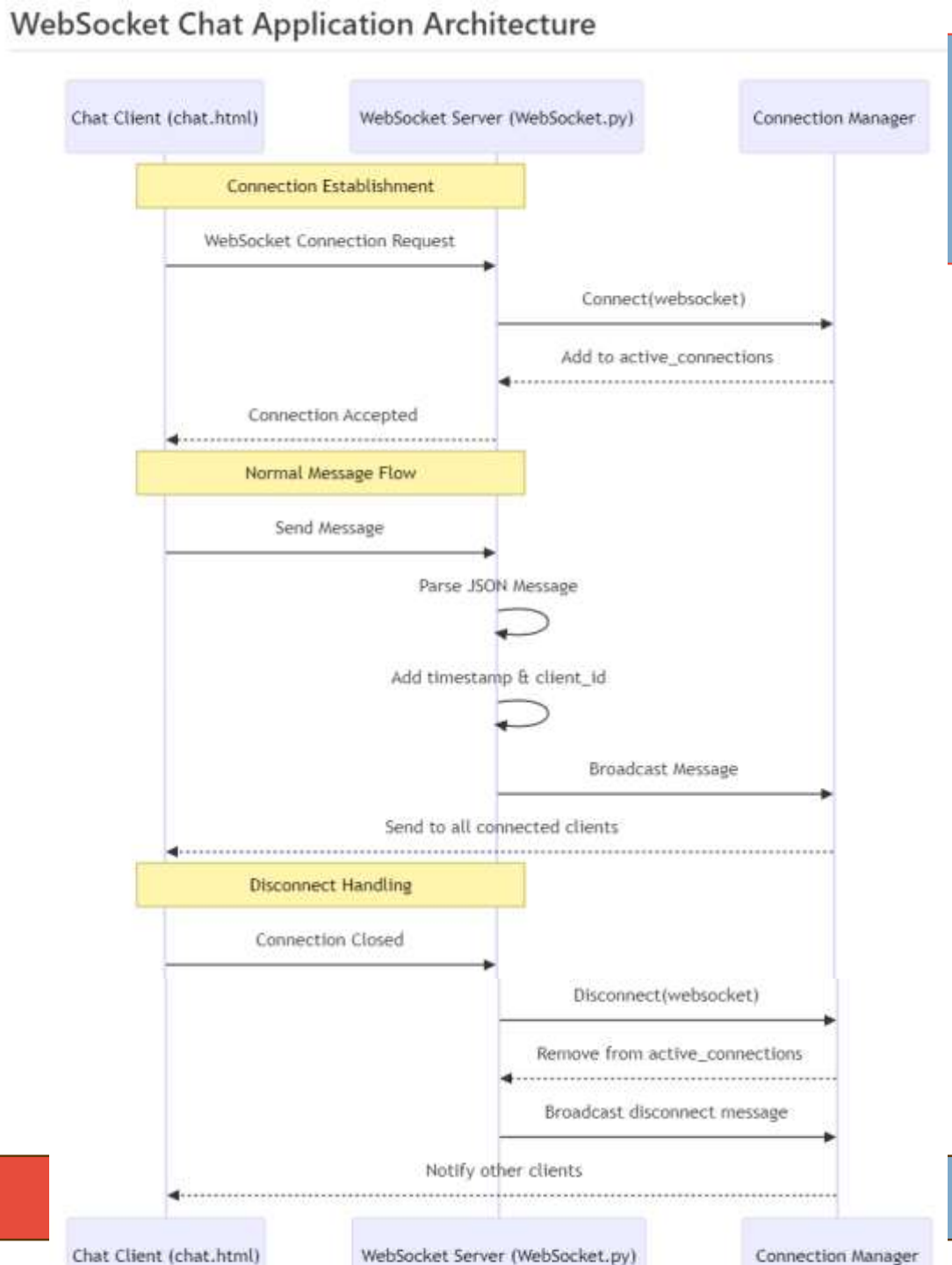
class ConnectionManager

WebSocket.py | index.html

WebSocket.py > manager

```python
# main.py
from fastapi import FastAPI, WebSocket, WebSocketDisconnect
from typing import List
import json
from datetime import datetime

app = FastAPI()

class ConnectionManager:
    def __init__(self):
        self.active_connections: List[WebSocket] = []
        self.connection_count = 0

    async def connect(self, websocket: WebSocket):
        await websocket.accept()
        self.active_connections.append(websocket)
        self.connection_count += 1
        print(f"New connection established. Total connections: {self.connection_count}")

    def disconnect(self, websocket: WebSocket):
        self.active_connections.remove(websocket)
        self.connection_count -= 1
        print(f"Connection closed. Total connections: {self.connection_count}")

    async def broadcast(self, message: str):
        print(f"Broadcasting message: {message}")
        for connection in self.active_connections:
            await connection.send_text(message)

manager = ConnectionManager()

```

sequence Diagram



Connection Manager Class Implementation

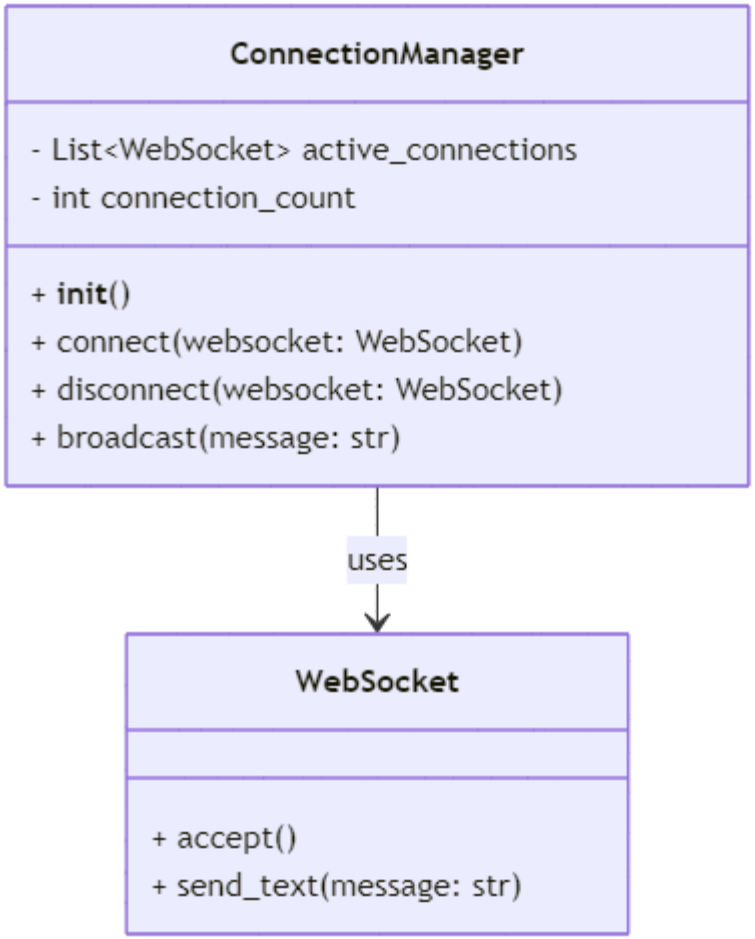
- Key Components:

- Maintains list of active connections
- Handles connection acceptance
- Manages disconnections
- Implements broadcast functionality

- Core Methods:

- connect(): Accept and store new connections
- disconnect(): Remove connections
- broadcast(): Send messages to all clients

class Diagram



WebSocket Endpoint Implementation

```python
@app.websocket("/ws/{client_id}")
async def websocket_endpoint(websocket: WebSocket, client_id: str):
    await manager.connect(websocket)
    print(f"Client {client_id} connected")

    try:
        while True:
            data = await websocket.receive_text()

            try:
                message_data = json.loads(data)
                timestamp = datetime.now().strftime("%H:%M:%S")
                message_data.update({
                    "client_id": client_id,
                    "timestamp": timestamp
                })
                print(f"Received from client {client_id}: {message_data['message']}")
                await manager.broadcast(json.dumps(message_data))

            except json.JSONDecodeError:
                print(f"Received plain text from client {client_id}: {data}")
                await manager.broadcast(f"Client {client_id}: {data}")

    except WebSocketDisconnect:
        manager.disconnect(websocket)
        disconnect_message = f"Client #{client_id} left the chat"
        print(disconnect_message)
        await manager.broadcast(json.dumps({
            "client_id": "system",
            "message": disconnect_message,
            "timestamp": datetime.now().strftime("%H:%M:%S"),
            "type": "disconnect"
        }))
```

- **Endpoint Setup:**
  - Route definition with client_id
  - WebSocket connection handling
  - Async message processing
- **Key Features:**
  - Client identification
  - JSON message parsing
  - Continuous message loop
  - Disconnect handling
  - Error management

```
Received from client 7w8cwb: Hi Dr. Mohammed, How are you ?
Broadcasting message: {"message": "Hi Dr. Mohammed, How are you ?", "client_id": "7w8cwb", "timestamp": "22:27:39"}
Received from client 7w8cwb: We are in the ACP Class using WebSocket and FastAPI
Broadcasting message: {"message": "We are in the ACP Class using WebSocket and FastAPI", "client_id": "7w8cwb", "timestamp": "22:28:22"}
```

Client Interface HTML

Chat application

https://www.youtube.com/watch?v=lQ7s_vJ7naM

https://youtu.be/CdENaxFNQiY

https://github.com/DrMohammedhbi/WebSockets-in-FastAPI/tree/main

```
<body>
    <div id="connectionStatus"></div>
    <div id="messages"></div>
    <div class="input-container">
        <input type="text" id="messageText" placeholder="Type your message...">
        <button onclick="sendMessage()">Send</button>
    </div>
```

https://fastapi.tiangolo.com/advanced/testing-websockets/

Client-Side JavaScript Implementation

- WebSocket Connection:

- Establish connection with unique client ID
- Handle connection events (open, close, error)
- Process incoming messages

```
function sendMessage() {
    const messageInput = document.getElementById('messageText');
    const message = messageInput.value.trim();
    if (message) {
        ws.send(JSON.stringify({
            message: message
        }));
        messageInput.value = '';
    }
}
```

- Message Handling:

- Send JSON formatted messages
- Auto-scroll message display
- Support for Enter key submission
- Real-time message updates

```
document.getElementById('messageText').addEventListener('keypress', function(e) {
    if (e.key === 'Enter') {
        sendMessage();
    }
});
```

Client-Side JavaScript Implementation

```
<script>
    const clientId = Math.random().toString(36).substring(7);
    const ws = new WebSocket(`ws://localhost:8000/ws/${clientId}`);
    const messages = document.getElementById('messages');
    const connectionStatus = document.getElementById('connectionStatus');

    function updateConnectionStatus(connected) {
        connectionStatus.textContent = connected ? 'Connected' : 'Disconnected';
        connectionStatus.className = connected ? 'connected' : 'disconnected';
    }

    ws.onopen = function() {
        updateConnectionStatus(true);
    };

    ws.onmessage = function(event) {
        try {
            const data = JSON.parse(event.data);
            const messageDiv = document.createElement('div');

            if (data.type === 'disconnect') {
                messageDiv.className = 'message system';
                messageDiv.textContent = data.message;
            } else {
                messageDiv.className = `message ${data.client_id === clientId ? 'user' : 'other'}`;
                messageDiv.innerHTML = `
                    ${data.client_id === clientId ? 'You' : 'Client ' + data.client_id}:
                    ${data.message}
                    <span class="timestamp">${data.timestamp}</span>
                `;
            }

            messages.appendChild(messageDiv);
            messages.scrollTop = messages.scrollHeight;
        } catch (e) {
            const messageDiv = document.createElement('div');
            messageDiv.className = 'message system';
            messageDiv.textContent = event.data;
            messages.appendChild(messageDiv);
        }
    };
```

Best Practices and Error Handling

Best Practices

- Use connection manager
- Implement proper error handling
- Handle disconnections gracefully
- Maintain clean connection state

Error Handling

- JSON decode errors
- Connection failures
- Client disconnections
- Message broadcast failures

```
ws.onerror = function() {
    updateConnectionStatus(false);
    const messageDiv = document.createElement('div');
    messageDiv.className = 'message system';
    messageDiv.textContent = 'Error occurred';
    messages.appendChild(messageDiv);
};
```

HTML: An Unordered HTML List

Run

```
<html>
<body>

<h2>An Unordered HTML List</h2>

<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>

<h2>An Ordered HTML List</h2>

<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>

</body>
</html>
```

An Unordered HTML List

- Coffee
- Tea
- Milk

An Ordered HTML List

1. Coffee
2. Tea
3. Milk

Ordered HTML List

An ordered list starts with the `<ol>` tag. Each list item starts with the `<li>` tag.

The list items will be marked with numbers by default:

Example

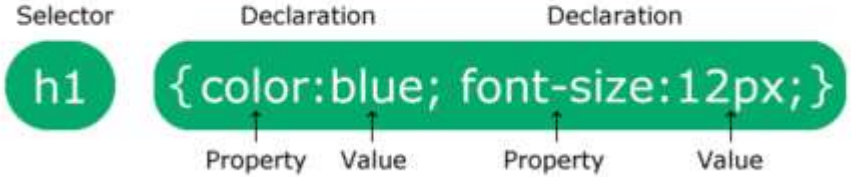
```
<ol>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ol>
```

CSS

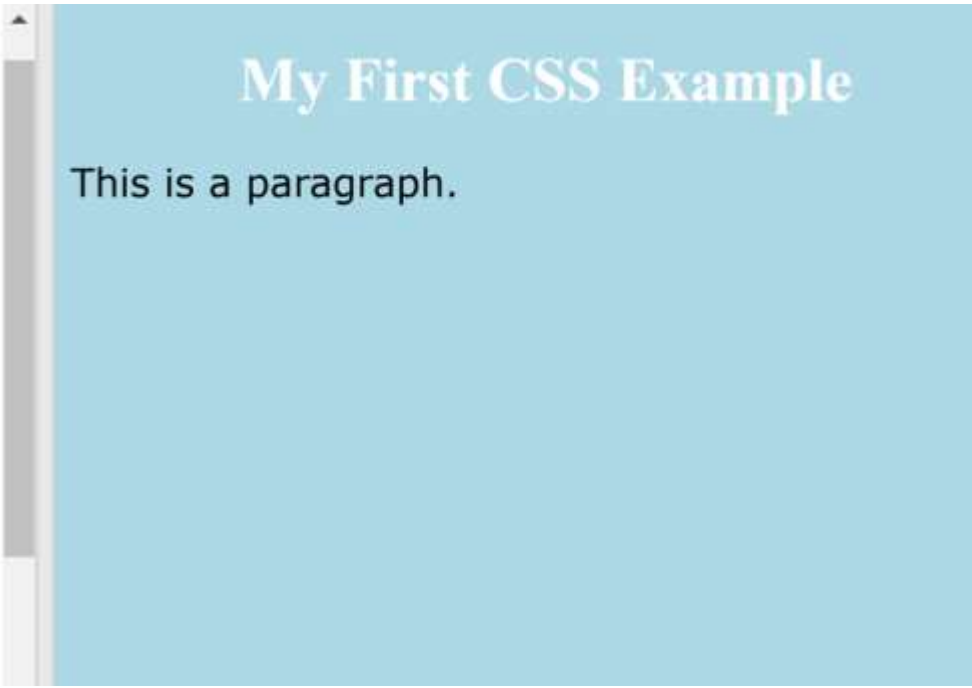
```
<style>
    body {
        font-family: Arial, sans-serif;
        max-width: 800px;
        margin: 20px auto;
        padding: 20px;
        background-color: #f5f5f5;
    }

    #messages {
        height: 400px;
        overflow-y: scroll;
        border: 1px solid #ddd;
        padding: 15px;
        margin-bottom: 20px;
        background-color: white;
        border-radius: 5px;
        box-shadow: 0 2px 4px rgba(0,0,0,0.1);
    }
```

CSS

CSS Syntax

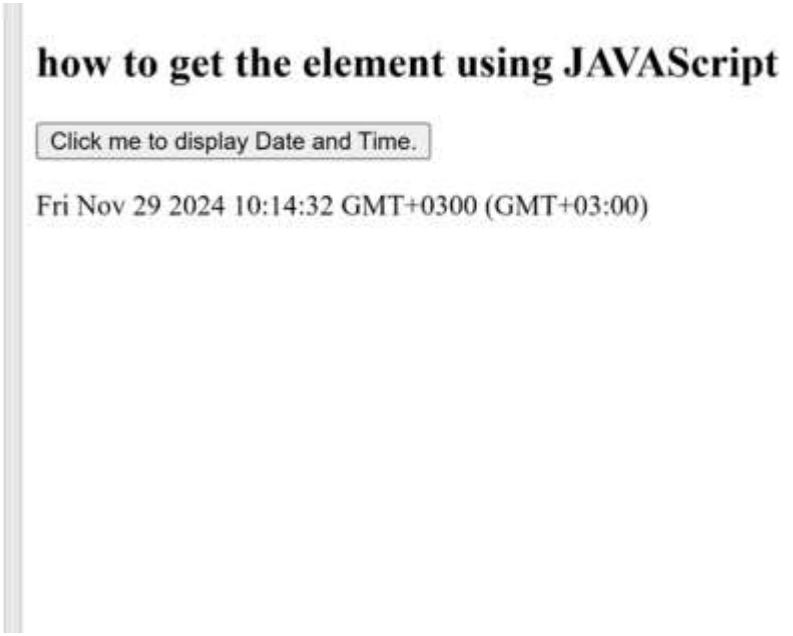



```
<html>
<head>
<style>
body {
  background-color: lightblue;
}

h1 {
  color: white;
  text-align: center;
}

p {
  font-family: verdana;
  font-size: 20px;
}
</style>
</head>
<body>
```



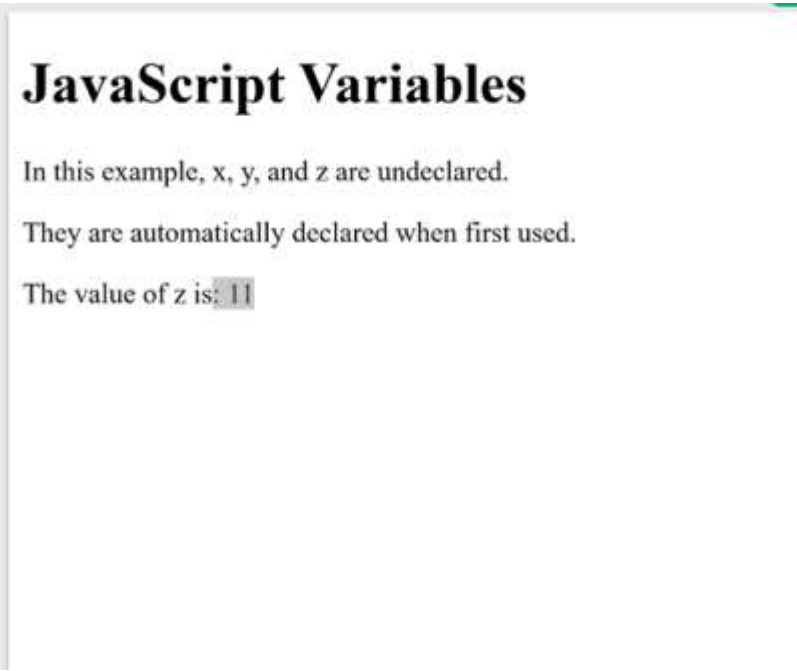
JAVAScript

```
<!DOCTYPE html>
<html>
<body>

<h2>how to get the HTML p element using JAVAScript</h2>

<button type="button"
onclick="document.getElementById('demo').innerHTML = Date()">
Click me to display Date and Time.</button>

<p id="demo"></p>

</body>
</html>
```



JAVAScript example

```
<!DOCTYPE html>
<html>
<body>
<h1>JavaScript Variables</h1>

<p>In this example, x, y, and z are undeclared.</p>
<p>They are automatically declared when first used.</p>

<p id="demo"></p>

<script>
x = 5;
y = 6;
z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>

</body>
</html>
```



JAVAScript code

```
<script>
    const clientId = Math.random().toString(36).substring(7);
    const ws = new WebSocket(`ws://localhost:8000/ws/${clientId}`);
    const messages = document.getElementById('messages');
    const connectionStatus = document.getElementById('connectionStatus');

    function updateConnectionStatus(connected) {
        connectionStatus.textContent = connected ? 'Connected' : 'Disconnected';
        connectionStatus.className = connected ? 'connected' : 'disconnected';
    }

    ws.onopen = function() {
        updateConnectionStatus(true);
    };

    ws.onmessage = function(event) {
        try {
            const data = JSON.parse(event.data);
            const messageDiv = document.createElement('div');

            if (data.type === 'disconnect') {
                messageDiv.className = 'message system';
                messageDiv.textContent = data.message;
            } else {
                messageDiv.className = `message ${data.client_id === clientId ? 'user' : 'other'}`;
```

References

book

https://www.w3schools.com/css/default.asp

https://www.w3schools.com/html/html_lists.asp

https://www.w3schools.com/js/default.asp

8

Defining WebSockets for Two-Way Interactive Communication in FastAPI

HTTP is a simple yet powerful technique for sending data to and receiving data from a server. As we've seen, the principles of request and response are at the core of this protocol: when developing our API, our goal is to process the incoming request and build a response for the client. Thus, in order to get data from the server, the client always has to initiate a request first. In some contexts, however, this may not be very convenient. Imagine a typical chat application: when a user receives a new message, we would like them to be notified immediately by the server. Working only with HTTP, we would have to make requests every second to check whether new messages had arrived, which would be a massive waste of resources. This is why a new protocol has emerged: **WebSocket**. The goal of this protocol is to open a communication channel between a client and a server so that they can exchange data in real time, in both directions.

In this chapter, we're going to cover the following main topics:

Good Luck