Video Conferencing Web App

Written By: Mr. Shubham Oulkar

Last Revised Date: 17 – Aug – 2023

Document Control:

Version	Date	Author	Comments
1.0	23 – Aug – 2023	Shubham Oulkar	Architecture defined

Reviews:

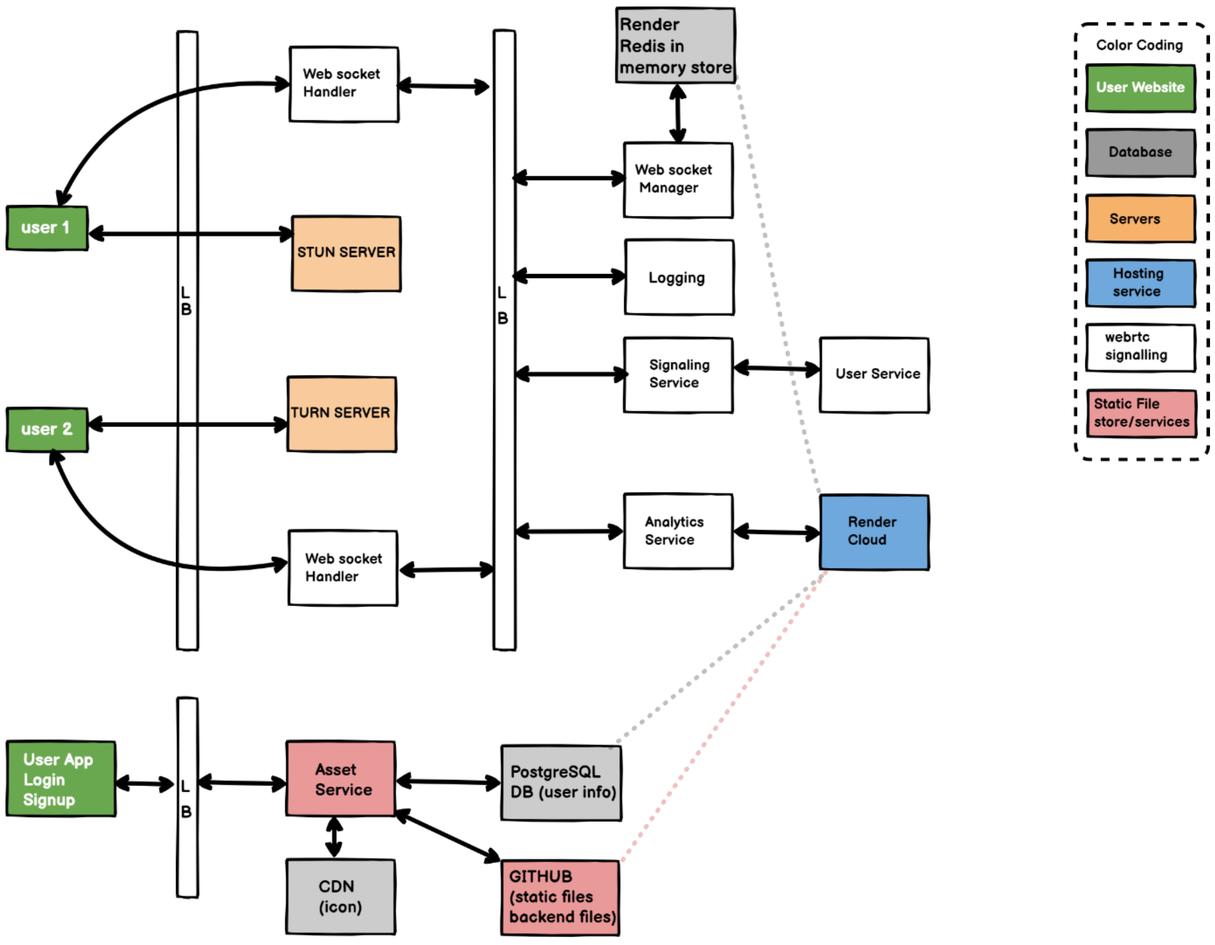
Version	Date	Author	Comments

Contents

Introduction

Designing a low-level architecture for a video calling web app using WebRTC, WebSockets, and Django involves integrating the real-time communication capabilities of WebRTC and WebSockets with the Django framework for backend management. This architecture leverages Django's capabilities for backend management and real-time communication through the channels library, while integrating WebRTC for video and audio streaming and WebSockets for signaling. Adapt this outline to your specific requirements and consider additional features, security measures, and optimizations as needed.

Architecture:



3.1 User Interface (UI):

Designing a user interface (UI) for a video calling web app with screen sharing involves creating a visually appealing, intuitive, and functional interface that allows users to interact seamlessly with the video call and screen sharing features. Here's a detailed breakdown of the UI components and their functionality:

1. Landing Page
   - Landing page provide information about the app and it's features
   - It includes a "Sign Up" and "Log In" button for users to access the app.
2. Authentication and Registration:
   - User should be able to create accounts or log in using their credentials.
   - Use HTML forms with post method, Django have build in csrf protection.
   - Provided password recovery options, such as "Forgot Password".
3. Page to create channel
   Upon successful login users are directed to the create channel page. User can create their own channel.
4. Video Calling page
   - Call Controls : mute/unmute audio, turn on/off video, hangup call, screen sharing button
   - User can see remote peer name as well as himself on the screen.
   - Internet connection status with channel name.
5. Responsive Design : UI is responsive and adapts to various screen sizes (destop, tablet, mobile).
6. Accessibility:
   - Make the UI accessible to users with disabilities by adhering to accessibility guidelines.
   - Provide keyboard shortcuts and screen reader compatibility.
7. Error Handling:
   Display clear error messages for network issues, failed connections, or other errors. Offer helpful tips for troubleshooting.
8. Users can log out securely when they're done using the app.

3.2 Signalling Server:

It facilitates the initial connection and communication between peers, allowing them to exchange information required to establish a direct connection for media streaming. Here's a detailed explanation of the signaling server's role, components, and operations:

1. Role of the Signaling Server: The primary role of the signaling server is to coordinate the following aspects of the WebRTC communication: Session Initiation: The signaling server helps establish a connection between peers who want to communicate. It assists in finding and connecting participants. ICE Candidate Exchange: The server relays information about network connectivity (ICE candidates) to help peers find the most efficient communication path. Offer and Answer Exchange: Peers exchange session descriptions (offers and answers) to establish the WebRTC connection parameters. Session Termination: The signaling server can also handle the termination of sessions or calls.
2. Components of the Signaling Server: The signaling server consists of several components to manage user connections and relay messages:
   WebSocket Server: Utilizes WebSockets to enable bidirectional communication between clients (web browsers) and the server. This is often used for real-time message exchange.
   Message Handling Logic: The server implements logic to handle different types of messages, such as offers, answers, ICE candidates, and termination signals.

3.3 WebRTC Connection Components:

WebRTC connection components are fundamental building blocks that facilitate peer-to-peer communication. Here's a detailed explanation of the key components involved in establishing and managing a WebRTC connection:
1. MediaStream API: The `MediaStream` API allows access to audio and video media sources, such as the user's camera and microphone. It provides a way to capture and manage audio and video streams for transmission over the WebRTC connection.
2. RTCPeerConnection: The `RTCPeerConnection` API is at the core of WebRTC, handling the establishment and management of the peer-to-peer connection. It enables audio and video communication, as well as data channels for non-media communication. The key functionalities of the `RTCPeerConnection` include,
Offer and Answer Exchange: Peers exchange session descriptions (offers and answers) that include information about supported media codecs and connection configurations.

ICE Candidate Exchange: ICE (Interactive Connectivity Establishment) candidates represent possible public IP addresses and ports for establishing a connection. Peers exchange these candidates to find the optimal connection path.

Negotiation of Connection Parameters: Peers negotiate connection parameters such as audio and video codecs, bandwidth, and other settings to ensure compatibility.

Media Streaming: The `RTCPeerConnection` handles the transmission and reception of audio and video streams between peers.

Connection Establishment and Termination: It establishes a direct connection between peers and manages the termination of connections (signalling servers are not used for video, audio calling).

ICE (Interactive Connectivity Establishment): ICE is a technique used by WebRTC to establish a connection even when both peers are behind firewalls or NATs (Network Address Translators). It gathers network information (ICE candidates) and uses various protocols to determine the best communication path.

STUN (Session Traversal Utilities for NAT): STUN servers help discover a client's public IP address and port by sending a simple request. This information is useful for establishing peer-to-peer connections through NATs.

TURN (Traversal Using Relays around NAT): If direct peer-to-peer communication is not possible due to restrictive network conditions(symmetric NAT), a TURN server can act as a relay to route media traffic between peers. It's a fallback mechanism for cases when direct communication is blocked.

3.4 Django:

User Authentication:

Django provides a robust authentication system out of the box. You can use it to manage user accounts, login, registration, and security. The authentication system handles tasks such as password hashing, session management, and user roles.

Database Models:

Define database models to represent users. Use Django's Object-Relational Mapping (ORM) to create and interact with these models using Python code. The ORM abstracts away most of the SQL operations.

Views and Controllers:

Views in Django handle user requests and return responses, which can be HTML templates, JSON, or other formats. Controllers (also known as views) handle the business logic associated with these requests. They interact with models, fetch data, and pass it to templates or APIs.

Signaling with Channels:

To implement WebSockets for real-time communication, use Django Channels, an extension that enables asynchronous and real-time capabilities. Channels allow you to handle WebSocket connections, receive and send messages, and manage channels for different users and sessions.

Session Management:

Django provides session management to maintain user sessions across requests. You can customize session settings, including expiration times and session data storage.

Error Handling with logging:

Implement error handling mechanisms to gracefully manage exceptions and unexpected situations. Return appropriate error responses and messages to the client.

URL Routing:

Define URL patterns using Django's URL routing system to map URLs to specific views or API endpoints.

Security:

Ensure your Django backend is secure. Implement CSRF protection, input validation, and consider implementing CORS (Cross-Origin Resource Sharing) if your frontend and backend are on different domains.

Testing:

Write unit tests and integration tests to ensure the reliability and correctness of your backend. Django provides tools for creating and running tests.

3.4 Database: This project use PostgreSQL database from render environment.

3.5 Deployment: Deployed on render because this platform is free to use. Also it provides internal PostgreSQL , redis support. So that we can minimise signalling time for a system. Render extremely simple dashboard and great git integration allow you to get an app upand running in minutes. It has native support for Python. Render also provide basic analytics chart like connection time, memory usage, Bandwidth usage, CPU time and many more.