Video Conferencing Application Documentation

Project Overview

This document outlines the architecture, features, and implementation details for a Django-based video conferencing application. The application enables real-time communication between users through video, audio, chat messaging, and screen sharing functionalities.

Technology Stack

Core Technologies

Feature	Technology
Backend	Django (Python)
Frontend	HTML, CSS, JavaScript, Bootstrap or Tailwind
Real-Time Communication	WebRTC (for peer-to-peer video/audio streaming)
Signaling Server	Django Channels (WebSocket support) or Node.js (optional)
Chat/Messaging	Django Channels or Socket.IO
Authentication	Django Auth / JWT (for APIs)
Database	PostgreSQL / MySQL / SQLite
Media Handling	WebRTC APIs, STUN/TURN servers for connectivity

Optional/Advanced Features

Feature	Tools
Screen Sharing	WebRTC APIs
Group Calls	WebRTC (SFU like Jitsi, Janus, or mediasoup if scaling)
File Sharing	Django File Uploads + Media Storage
Notifications	Web Push API / Django Signals
Deployment	VPS/Cloud (e.g., Heroku, DigitalOcean, AWS)
Domain & SSL	For secure media stream via HTTPS

System Architecture

High-Level Architecture

```
┌─────────────────┐     ┌─────────────────┐     ┌────────────────┐
│   Client Side   │◄────┤  Django Server  │◄────┤    Database    │
│   (Browser)     │     │   + Channels    │     │  (PostgreSQL)  │
└────────┬────────┘     └────────┬────────┘     └────────────────┘
         │                       │
         │                       │
         │                       │
     WebRTC P2P           ┌──────▼──────┐
         │                │  STUN/TURN  │
         ▼                │   Servers   │
┌─────────────────┐       └─────────────┘
│  Other Clients  │
│   (Browsers)    │
└─────────────────┘
```

Data Flow

1. Users authenticate through Django's authentication system
2. Django Channels establishes WebSocket connections for signaling
3. WebRTC establishes peer connections between clients
4. STUN/TURN servers assist with NAT traversal when needed
5. Real-time data (video, audio, chat) flows directly between peers through WebRTC

Feature Specifications

1. User Authentication System

Registration Page Components

- Full Name (Text input)
- Username (Text input)
- Email (Email input)
- Phone (Number input)
- Password (Password input)
- Confirm Password (Password input)
- Select Profile (Dropdown or File Upload for picture)
- Send OTP Button (Triggers email OTP)
- Enter OTP (Input field)
- Complete Registration Button

Registration Process Flow

1. User fills out registration form
2. System validates input (email format, password strength, etc.)
3. System sends OTP to provided email
4. User verifies identity with OTP
5. Account is created and stored in database
6. User is redirected to login page

Login Page Components

- Email or Username (Text input)
- Password (Password input)
- Login Button
- Forgot Password link (Optional)

2. User Dashboard

Components

- User information display (Full Name, Username, Email, Phone, Profile Picture)
- Edit Profile Button
- Join Meeting / Host Meeting options
- Logout Button

Dashboard Features

- Personal meeting ID display
- Meeting history (past meetings)
- Scheduled meetings (if implementing calendar feature)
- Quick start meeting button
- Join meeting with code input

3. Meeting Interface

Core Components

- Video Preview Window
- Microphone Toggle Button (Mute/Unmute)

- Camera Toggle Button (On/Off)
- Screen Share Button
- Audio Share Button
- Chat Panel / Messages
- Participants List
- Leave Meeting Button

Advanced Meeting Features

- Room creation with custom URLs
- Meeting password protection
- Waiting room functionality
- Host controls (mute all, remove participant)
- Recording options (if implementing)
- Virtual background (optional enhancement)

Technical Implementation

Backend Implementation (Django)

Models

```python
# Example models.py structure

from django.db import models
from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    phone_number = models.CharField(max_length=15, blank=True)
    profile_picture = models.ImageField(upload_to='profile_pics/', blank=True)

class Meeting(models.Model):
    meeting_id = models.CharField(max_length=20, unique=True)
    host = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=200)
    password = models.CharField(max_length=50, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)

class MeetingParticipant(models.Model):
    meeting = models.ForeignKey(Meeting, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    joined_at = models.DateTimeField(auto_now_add=True)
    left_at = models.DateTimeField(null=True, blank=True)

class ChatMessage(models.Model):
    meeting = models.ForeignKey(Meeting, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    timestamp = models.DateTimeField(auto_now_add=True)
```

Django Channels Configuration

python

```python
# Example channels setup

# settings.py
INSTALLED_APPS = [
    # ...
    'channels',
    'meeting',
]

ASGI_APPLICATION = 'project.asgi.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            "hosts": [('127.0.0.1', 6379)],
        },
    },
}

# asgi.py
import os
from channels.routing import ProtocolTypeRouter, URLRouter
from django.core.asgi import get_asgi_application
from channels.auth import AuthMiddlewareStack
from meeting.routing import websocket_urlpatterns

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'project.settings')

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter(
            websocket_urlpatterns
        )
    ),
})
```

WebRTC Signaling Implementation

python

```python
# Example consumer for WebRTC signaling

import json
from channels.generic.websocket import AsyncWebsocketConsumer

class SignalingConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
        self.room_group_name = f'meeting_{self.room_name}'

        # Join room group
        await self.channel_layer.group_add(
            self.room_group_name,
            self.channel_name
        )
        await self.accept()

    async def disconnect(self, close_code):
        # Leave room group
        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

    # Receive WebRTC signal from WebSocket
    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message_type = text_data_json['type']

        # Forward the signal to the room group
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'signal_message',
                'message': text_data_json,
                'sender_channel_name': self.channel_name
            }
        )

    # Receive message from room group
    async def signal_message(self, event):
        message = event['message']
```

```python
        # Don't send the message back to the sender
        if self.channel_name != event['sender_channel_name']:
            # Send message to WebSocket
            await self.send(text_data=json.dumps(message))
```

Frontend Implementation

WebRTC Setup (JavaScript)

javascript

```javascript
// Example WebRTC connection setup

// Configuration with STUN/TURN servers
const peerConnectionConfig = {
    iceServers: [
        { urls: 'stun:stun.l.google.com:19302' },
        {
            urls: 'turn:your-turn-server.com:3478',
            username: 'username',
            credential: 'credential'
        }
    ]
};

let localStream;
let peerConnections = {};
let socket;

// Setup WebSocket connection to signaling server
function connectSignalingServer(roomId) {
    socket = new WebSocket(`ws://${window.location.host}/ws/meeting/${roomId}/`);

    socket.onmessage = function(e) {
        const data = JSON.parse(e.data);

        switch(data.type) {
            case 'offer':
                handleOffer(data);
                break;
            case 'answer':
                handleAnswer(data);
                break;
            case 'ice-candidate':
                handleIceCandidate(data);
                break;
            case 'user-joined':
                createPeerConnection(data.userId);
                break;
            case 'user-left':
                closePeerConnection(data.userId);
                break;
        }
    };
```

```
}

// Set up media stream and start connection
async function setupMediaStream() {
    try {
        localStream = await navigator.mediaDevices.getUserMedia({
            video: true,
            audio: true
        });

        // Display local video
        document.getElementById('local-video').srcObject = localStream;

        // Notify server that user has joined
        socket.send(JSON.stringify({
            type: 'join',
            roomId: currentRoomId
        }));
    } catch (error) {
        console.error('Error accessing media devices:', error);
    }
}

// Create a peer connection with another user
function createPeerConnection(userId) {
    const peerConnection = new RTCPeerConnection(peerConnectionConfig);
    peerConnections[userId] = peerConnection;

    // Add local stream tracks to the connection
    localStream.getTracks().forEach(track => {
        peerConnection.addTrack(track, localStream);
    });

    // Handle ICE candidates
    peerConnection.onicecandidate = event => {
        if (event.candidate) {
            socket.send(JSON.stringify({
                type: 'ice-candidate',
                candidate: event.candidate,
                targetUserId: userId
            }));
        }
    };
```

```javascript
    // Handle incoming tracks (remote video/audio)
    peerConnection.ontrack = event => {
        // Create or get remote video element
        let remoteVideo = document.getElementById(`remote-video-${userId}`);
        if (!remoteVideo) {
            remoteVideo = document.createElement('video');
            remoteVideo.id = `remote-video-${userId}`;
            remoteVideo.autoplay = true;
            document.getElementById('videos-container').appendChild(remoteVideo);
        }
        remoteVideo.srcObject = event.streams[0];
    };

    // Create and send offer
    peerConnection.createOffer()
        .then(offer => peerConnection.setLocalDescription(offer))
        .then(() => {
            socket.send(JSON.stringify({
                type: 'offer',
                sdp: peerConnection.localDescription,
                targetUserId: userId
            }));
        })
        .catch(error => console.error('Error creating offer:', error));

    return peerConnection;
}

// Handle incoming offer
function handleOffer(data) {
    const peerConnection = new RTCPeerConnection(peerConnectionConfig);
    peerConnections[data.userId] = peerConnection;

    // Add local stream tracks to the connection
    localStream.getTracks().forEach(track => {
        peerConnection.addTrack(track, localStream);
    });

    // Handle ICE candidates
    peerConnection.onicecandidate = event => {
        if (event.candidate) {
            socket.send(JSON.stringify({
                type: 'ice-candidate',
                candidate: event.candidate,
```

```
                targetUserId: data.userId
            }));
        }
    };

    // Handle incoming tracks (remote video/audio)
    peerConnection.ontrack = event => {
        // Create or get remote video element
        let remoteVideo = document.getElementById(`remote-video-${data.userId}`);
        if (!remoteVideo) {
            remoteVideo = document.createElement('video');
            remoteVideo.id = `remote-video-${data.userId}`;
            remoteVideo.autoplay = true;
            document.getElementById('videos-container').appendChild(remoteVideo);
        }
        remoteVideo.srcObject = event.streams[0];
    };

    // Set remote description and create answer
    peerConnection.setRemoteDescription(new RTCSessionDescription(data.sdp))
        .then(() => peerConnection.createAnswer())
        .then(answer => peerConnection.setLocalDescription(answer))
        .then(() => {
            socket.send(JSON.stringify({
                type: 'answer',
                sdp: peerConnection.localDescription,
                targetUserId: data.userId
            }));
        })
        .catch(error => console.error('Error handling offer:', error));
}

// Handle incoming answer
function handleAnswer(data) {
    const peerConnection = peerConnections[data.userId];
    if (peerConnection) {
        peerConnection.setRemoteDescription(new RTCSessionDescription(data.sdp))
            .catch(error => console.error('Error handling answer:', error));
    }
}

// Handle incoming ICE candidate
function handleIceCandidate(data) {
    const peerConnection = peerConnections[data.userId];
```

```
    if (peerConnection) {
        peerConnection.addIceCandidate(new RTCIceCandidate(data.candidate))
            .catch(error => console.error('Error adding ICE candidate:', error));
    }
}

// Close peer connection when user leaves
function closePeerConnection(userId) {
    if (peerConnections[userId]) {
        peerConnections[userId].close();
        delete peerConnections[userId];

        // Remove remote video element
        const remoteVideo = document.getElementById(`remote-video-${userId}`);
        if (remoteVideo) {
            remoteVideo.srcObject = null;
            remoteVideo.remove();
        }
    }
}
```

UI Component Implementation Examples

Meeting Controls UI

html

```
<div class="meeting-controls">
    <button id="toggle-audio" class="control-btn">
        <i class="fas fa-microphone"></i>
    </button>
    <button id="toggle-video" class="control-btn">
        <i class="fas fa-video"></i>
    </button>
    <button id="share-screen" class="control-btn">
        <i class="fas fa-desktop"></i>
    </button>
    <button id="toggle-chat" class="control-btn">
        <i class="fas fa-comments"></i>
    </button>
    <button id="leave-meeting" class="control-btn leave">
        <i class="fas fa-phone-slash"></i>
    </button>
</div>

<script>
    // Toggle audio
    document.getElementById('toggle-audio').addEventListener('click', () => {
        const audioTrack = localStream.getAudioTracks()[0];
        if (audioTrack) {
            audioTrack.enabled = !audioTrack.enabled;
            document.getElementById('toggle-audio').innerHTML =
                audioTrack.enabled ? '<i class="fas fa-microphone"></i>' : '<i class="fas fa-mi
        }
    });

    // Toggle video
    document.getElementById('toggle-video').addEventListener('click', () => {
        const videoTrack = localStream.getVideoTracks()[0];
        if (videoTrack) {
            videoTrack.enabled = !videoTrack.enabled;
            document.getElementById('toggle-video').innerHTML =
                videoTrack.enabled ? '<i class="fas fa-video"></i>' : '<i class="fas fa-video-s
        }
    });

    // Share screen
    document.getElementById('share-screen').addEventListener('click', async () => {
        try {
            const screenStream = await navigator.mediaDevices.getDisplayMedia({
```

```
                video: true
            });

            // Replace video track with screen share track
            const videoTrack = localStream.getVideoTracks()[0];
            if (videoTrack) {
                const sender = peerConnections[Object.keys(peerConnections)[0]]
                    .getSenders().find(s => s.track.kind === videoTrack.kind);
                sender.replaceTrack(screenStream.getVideoTracks()[0]);
            }

            // When user stops screen sharing
            screenStream.getVideoTracks()[0].onended = () => {
                const sender = peerConnections[Object.keys(peerConnections)[0]]
                    .getSenders().find(s => s.track.kind === 'video');
                sender.replaceTrack(localStream.getVideoTracks()[0]);
            };
        } catch (error) {
            console.error('Error sharing screen:', error);
        }
    });
</script>
```

Chat Interface

html

```html
<div class="chat-panel" id="chat-panel">
    <div class="chat-messages" id="chat-messages">
        <!-- Messages will be inserted here -->
    </div>
    <div class="chat-input">
        <input type="text" id="chat-message-input" placeholder="Type a message...">
        <button id="send-message-btn">Send</button>
    </div>
</div>

<script>
    // Send chat message
    document.getElementById('send-message-btn').addEventListener('click', () => {
        const messageInput = document.getElementById('chat-message-input');
        const message = messageInput.value.trim();

        if (message) {
            // Send message via WebSocket
            socket.send(JSON.stringify({
                type: 'chat-message',
                roomId: currentRoomId,
                message: message
            }));

            // Add message to chat (local display)
            addMessageToChat('You', message);

            // Clear input
            messageInput.value = '';
        }
    });

    // Add message to chat display
    function addMessageToChat(sender, message) {
        const messagesContainer = document.getElementById('chat-messages');
        const messageElement = document.createElement('div');
        messageElement.className = 'chat-message';
        messageElement.innerHTML = `
            <span class="message-sender">${sender}:</span>
            <span class="message-text">${message}</span>
        `;
        messagesContainer.appendChild(messageElement);
        messagesContainer.scrollTop = messagesContainer.scrollHeight;
```

```
    }

    // Handle incoming chat messages
    socket.addEventListener('message', (event) => {
        const data = JSON.parse(event.data);
        if (data.type === 'chat-message') {
            addMessageToChat(data.sender, data.message);
        }
    });
</script>
```

Deployment Guidelines

Server Requirements

- Python 3.8+
- Django 3.2+
- Redis (for Django Channels)
- PostgreSQL (recommended for production)
- STUN/TURN server (for NAT traversal)

Deployment Steps

1. Set up a VPS or cloud instance (AWS, DigitalOcean, etc.)
2. Install required dependencies
3. Configure PostgreSQL database
4. Set up Redis for Django Channels
5. Configure STUN/TURN servers
6. Set up a domain name and SSL certificate
7. Configure Nginx/Apache as a reverse proxy
8. Use Gunicorn or uWSGI as application server
9. Set up static file serving
10. Configure environment variables
11. Implement monitoring and error logging

Security Considerations

- Implement HTTPS for all connections

- Secure WebRTC traffic with DTLS
- Use secure WebSocket connections (WSS)
- Implement proper authentication and authorization
- Rate limit API endpoints
- Sanitize user inputs
- Implement proper error handling
- Regular security audits

Testing Strategy

Unit Testing

- Test Django models and views
- Test WebRTC connection logic
- Test authentication flows

Integration Testing

- Test WebSocket communication
- Test database interactions
- Test video/audio streaming

End-to-End Testing

- Test complete user flows
- Test multi-user scenarios
- Test on different browsers and devices

Performance Testing

- Test with multiple concurrent users
- Measure latency and bandwidth usage
- Test server resource utilization

Future Enhancements

- Mobile applications (iOS/Android)
- End-to-end encryption
- Virtual backgrounds

- Noise cancellation
- Meeting recordings
- Cloud storage integration
- Calendar integration
- Live transcription
- Breakout rooms
- Polls and Q&A features

Setup and Installation Guide

Local Development Setup

1. Clone the repository:

   bash

   ```bash
   git clone https://github.com/yourusername/video-conference-app.git
   cd video-conference-app
   ```

2. Create a virtual environment:

   bash

   ```bash
   python -m venv venv
   source venv/bin/activate  # On Windows: venv\Scripts\activate
   ```

3. Install dependencies:

   bash

   ```bash
   pip install -r requirements.txt
   ```

4. Set up environment variables:

   bash

   ```bash
   cp .env.example .env
   # Edit .env file with your settings
   ```

5. Run migrations:

   bash

   ```bash
   python manage.py migrate
   ```

6. Create a superuser:

```bash
python manage.py createsuperuser
```

7. Start Redis server (required for Channels):

```bash
redis-server
```

8. Run the development server:

```bash
python manage.py runserver
```

9. Access the application at `http://localhost:8000`

Production Deployment

For production deployment, follow the Deployment Guidelines section above and consider using tools like Docker for containerization and CI/CD pipelines for automated testing and deployment.