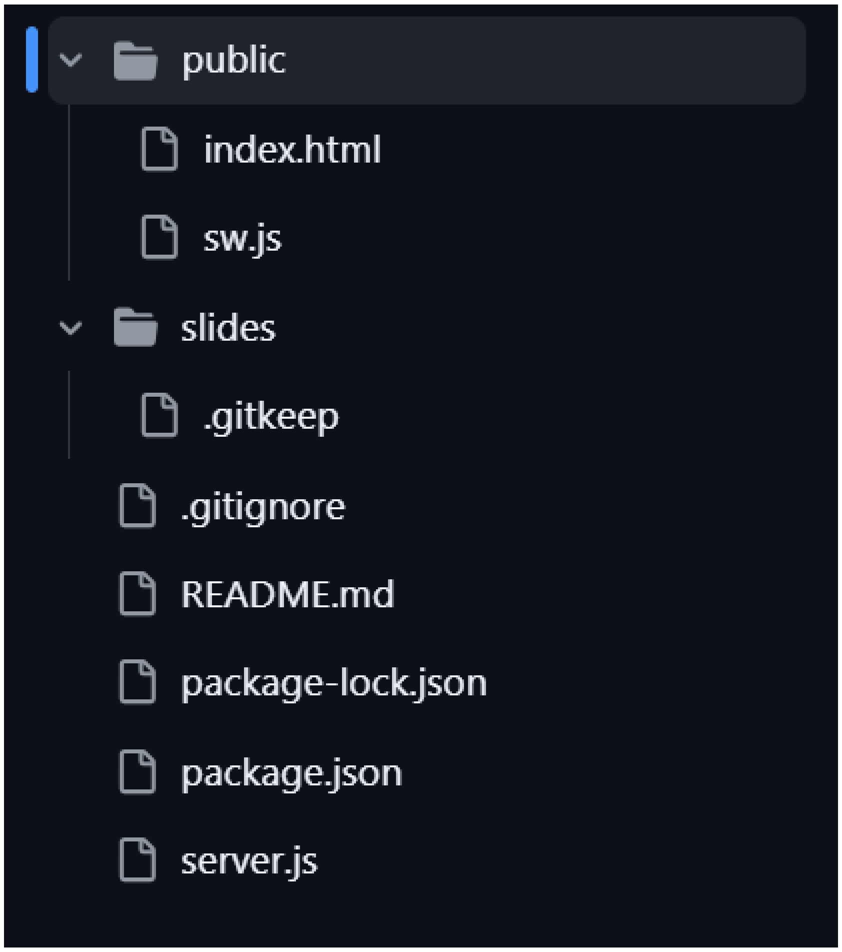
File structure:



```
public
    index.html
    sw.js
slides
    .gitkeep
.gitignore
README.md
package-lock.json
package.json
server.js
```

Public\index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Virtual Classroom - Smart India Hackathon</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.7.2/socket.io.js"></script>
  <style>
    * {
      margin: 0;
      padding: 0;
      box-sizing: border-box;
    }

    body {
      font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
    background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
    min-height: 100vh;
    color: #333;
}

.container {
    max-width: 1200px;
    margin: 0 auto;
    padding: 20px;
}

.header {
    background: rgba(255, 255, 255, 0.95);
    padding: 20px;
    border-radius: 15px;
    margin-bottom: 20px;
    backdrop-filter: blur(10px);
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
}

.join-form {
    display: flex;
    gap: 10px;
    align-items: center;
    flex-wrap: wrap;
}

.join-form input, .join-form select, .join-form button {
    padding: 12px;
    border: 2px solid #e1e5e9;
    border-radius: 8px;
    font-size: 16px;
}

.join-form button {
    background: #4CAF50;
    color: white;
    border: none;
    cursor: pointer;
    transition: background 0.3s;
}

.join-form button:hover {
    background: #45a049;
```

```css
}

.classroom {
    display: none;
    grid-template-columns: 2fr 1fr;
    gap: 20px;
    min-height: calc(100vh - 200px);
}

.main-content {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 20px;
    backdrop-filter: blur(10px);
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
    display: flex;
    flex-direction: column;
}

.slide-area {
    height: 400px;
    border: 2px dashed #ddd;
    border-radius: 10px;
    display: flex;
    align-items: center;
    justify-content: center;
    margin-bottom: 20px;
    background: #f9f9f9;
    position: relative;
    overflow: hidden;
}

.slide-content {
    max-width: 100%;
    max-height: 100%;
    object-fit: contain;
}

.slide-controls {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-bottom: 20px;
    flex-wrap: wrap;
    gap: 10px;
}

.slide-info {
    font-size: 14px;
    color: #666;
}

.teacher-controls {
    display: none;
    gap: 10px;
    flex-wrap: wrap;
}

.teacher-controls button, .audio-controls button {
    padding: 10px 15px;
    border: none;
    border-radius: 6px;
    cursor: pointer;
    transition: all 0.3s;
    font-size: 14px;
}

.teacher-controls .prev-btn {
    background: #ff6b6b;
    color: white;
}

.teacher-controls .prev-btn:hover {
    background: #ff5252;
}

.teacher-controls .next-btn {
    background: #4ecdc4;
    color: white;
}

.teacher-controls .next-btn:hover {
    background: #26c6da;
}

.upload-btn {
    background: #95a5a6;
    color: white;
```

```
}

.upload-btn:hover {
    background: #7f8c8d;
}

.audio-controls {
    margin-top: 10px;
    display: flex;
    gap: 10px;
    align-items: center;
    flex-wrap: wrap;
}

.start-audio-btn {
    background: #e74c3c;
    color: white;
}

.start-audio-btn:hover {
    background: #c0392b;
}

.stop-audio-btn {
    background: #95a5a6;
    color: white;
}

.stop-audio-btn:hover {
    background: #7f8c8d;
}

.sidebar {
    background: rgba(255, 255, 255, 0.95);
    border-radius: 15px;
    padding: 20px;
    backdrop-filter: blur(10px);
    box-shadow: 0 8px 32px rgba(0, 0, 0, 0.1);
    display: flex;
    flex-direction: column;
}

.participants {
    margin-bottom: 20px;
```

```css
}

.participants h3 {
   margin-bottom: 10px;
   color: #555;
}

.participant {
   padding: 8px;
   background: #f0f0f0;
   border-radius: 6px;
   margin-bottom: 5px;
   font-size: 14px;
}

.participant.teacher {
   background: #e8f5e8;
   font-weight: bold;
}

.chat {
   flex: 1;
   display: flex;
   flex-direction: column;
}

.chat h3 {
   margin-bottom: 10px;
   color: #555;
}

.chat-messages {
   flex: 1;
   border: 1px solid #ddd;
   border-radius: 8px;
   padding: 10px;
   overflow-y: auto;
   max-height: 300px;
   background: white;
   margin-bottom: 10px;
}

.message {
   margin-bottom: 10px;
```

```
    padding: 8px;
    border-radius: 6px;
    font-size: 14px;
}

.message.teacher {
    background: #e3f2fd;
    border-left: 3px solid #2196f3;
}

.message.student {
    background: #f3e5f5;
    border-left: 3px solid #9c27b0;
}

.message-header {
    font-weight: bold;
    color: #555;
    margin-bottom: 4px;
}

.chat-input {
    display: flex;
    gap: 10px;
}

.chat-input input {
    flex: 1;
    padding: 10px;
    border: 1px solid #ddd;
    border-radius: 6px;
}

.chat-input button {
    padding: 10px 15px;
    background: #2196f3;
    color: white;
    border: none;
    border-radius: 6px;
    cursor: pointer;
}

.chat-input button:hover {
    background: #1976d2;
```

```css
}

.status {
    position: fixed;
    top: 20px;
    right: 20px;
    padding: 10px 15px;
    border-radius: 6px;
    font-size: 14px;
    font-weight: bold;
    z-index: 1000;
}

.status.connected {
    background: #4CAF50;
    color: white;
}

.status.disconnected {
    background: #f44336;
    color: white;
}

.audio-status {
    padding: 10px;
    border-radius: 6px;
    text-align: center;
    font-size: 14px;
    min-width: 150px;
}

.audio-status.streaming {
    background: #c8e6c9;
    color: #2e7d32;
}

.audio-status.stopped {
    background: #ffcdd2;
    color: #c62828;
}

.hidden {
    display: none !important;
}
```

```css
.notification {
    position: fixed;
    top: 70px;
    right: 20px;
    padding: 10px 15px;
    border-radius: 6px;
    color: white;
    font-size: 14px;
    z-index: 1000;
    animation: slideIn 0.3s ease;
}

@keyframes slideIn {
    from {
        transform: translateX(100%);
        opacity: 0;
    }
    to {
        transform: translateX(0);
        opacity: 1;
    }
}

@media (max-width: 768px) {
    .classroom {
        grid-template-columns: 1fr;
        gap: 10px;
    }

    .container {
        padding: 10px;
    }

    .slide-area {
        height: 250px;
    }

    .join-form {
        flex-direction: column;
        align-items: stretch;
    }

    .slide-controls {
            flex-direction: column;
            align-items: stretch;
        }

        .teacher-controls {
            flex-direction: column;
        }

        .audio-controls {
            flex-direction: column;
        }
    }
  </style>
</head>
<body>
  <div class="container">
    <!-- Connection Status -->
    <div id="status" class="status disconnected">Disconnected</div>

    <!-- Join Form -->
    <div id="joinForm" class="header">
      <h1>🎓 Virtual Classroom for Rural Areas</h1>
      <p style="margin: 10px 0; color: #666;">Smart India Hackathon Prototype</p>
      <div class="join-form">
        <input type="text" id="nameInput" placeholder="Enter your name" required>
        <select id="roleSelect">
          <option value="student">Student</option>
          <option value="teacher">Teacher</option>
        </select>
        <button onclick="joinClassroom()">Join Classroom</button>
      </div>
    </div>

    <!-- Main Classroom Interface -->
    <div id="classroom" class="classroom">
      <!-- Main Content Area -->
      <div class="main-content">
        <div class="slide-area" id="slideArea">
          <div id="noSlideMessage">
            <p style="color: #999; font-size: 18px;">📋 Waiting for teacher to upload
slides...</p>
          </div>
          <img id="currentSlide" class="slide-content hidden" alt="Current Slide"
loading="lazy">
```

```
            </div>

            <div class="slide-controls">
              <div class="slide-info">
                <span id="slideInfo">Slide 0 of 0</span>
              </div>

              <div id="teacherControls" class="teacher-controls">
                <!-- Hidden file input -->
                <input
                  id="slideUpload"
                  type="file"
                  accept=".jpg,.jpeg,.png,.pdf,.pptx"
                  style="display: none;"
                  onchange="handleFileUpload(event)"
                />

                <!-- Upload button triggers input click -->
                <button class="upload-btn" onclick="triggerFileUpload()">
                  📤 Upload Slide
                </button>

                <!-- Navigation buttons -->
                <button class="prev-btn" onclick="previousSlide()">
                  ⬅️ Previous
                </button>
                <button class="next-btn" onclick="nextSlide()">
                  ➡️ Next
                </button>

                <!-- Resources: Upload (teacher only) -->
                <input id="resourceUpload" type="file" style="display:none"
onchange="handleResourceUpload(event)">
                <button class="upload-btn" onclick="triggerResourceUpload()">📦 Upload
Resource</button>
              </div>
            </div>

            <!-- Audio Controls -->
            <div class="audio-controls">
              <button id="startAudioBtn" class="start-audio-btn" onclick="startAudio()">
                🎤 Start Audio
              </button>
              <button id="stopAudioBtn" class="stop-audio-btn hidden" onclick="stopAudio()">
```

```
                🔇 Stop Audio
            </button>
            <div id="audioStatus" class="audio-status stopped">
                Audio: Stopped
            </div>
        </div>
    </div>

    <!-- Sidebar -->
    <div class="sidebar">
        <!-- Participants -->
        <div class="participants">
            <h3>👥 Participants (<span id="participantCount">0</span>)</h3>
            <div id="participantsList"></div>
        </div>

        <!-- Chat -->
        <div class="chat">
            <h3>💬 Chat</h3>
            <div id="chatMessages" class="chat-messages"></div>
            <div class="chat-input">
                <input type="text" id="chatInput" placeholder="Type a message..."
                       onkeypress="if(event.key==='Enter') sendMessage()">
                <button onclick="sendMessage()">Send</button>
            </div>
        </div>

        <!-- Resources Panel (visible to all) -->
        <div class="participants" style="margin-top: 16px;">
            <h3>📚 Resources</h3>
            <div id="resourcesList"></div>
        </div>
    </div>
  </div>
</div>
<script>
// Register Service Worker for offline caching
if ('serviceWorker' in navigator) {
  window.addEventListener('load', async () => {
    try {
      const reg = await navigator.serviceWorker.register('/sw.js');
      console.log('Service Worker registered:', reg.scope);
    } catch (e) {
      console.warn('Service Worker registration failed', e);
```

```
        }
    });
}
  // Global variables
let socket = null;
let currentUser = null;
let currentSlideNumber = 0;
let totalSlides = 0;
let slides = [];
let isAudioStreaming = false;
let localStream = null;
let peerConnections = new Map(); // Store multiple peer connections for students

// WebRTC configuration for low-bandwidth audio
const rtcConfiguration = {
    iceServers: [
        { urls: 'stun:stun.l.google.com:19302' },
        { urls: 'stun:stun1.l.google.com:19302' }
    ]
};

// Audio constraints optimized for low bandwidth
const audioConstraints = {
    audio: {
        echoCancellation: true,
        noiseSuppression: true,
        autoGainControl: true,
        sampleRate: 16000, // Lower sample rate for bandwidth efficiency
        sampleSize: 16,
        channelCount: 1 // Mono audio
    },
    video: false
};

// Initialize connection when page loads
window.onload = function() {
    console.log('Page loaded, initializing socket connection...');
    initializeSocket();
    setupChatEnterKey();
    // Fetch current resources and render + cache
    initResourcesIndex();
};

// Socket initialization
```

```
function initializeSocket() {
    socket = io();

    socket.on('connect', () => {
        updateStatus('connected', 'Connected');
        showConnectedMessage();
    });

    socket.on('disconnect', () => {
        updateStatus('disconnected', 'Disconnected');
        if (localStream) {
            localStream.getTracks().forEach(track => track.stop());
            localStream = null;
        }
        peerConnections.forEach(pc => pc.close());
        peerConnections.clear();
    });

    // Handle upload process events
    socket.on('upload-started', (data) => {
        console.log("Upload started:", data.filename);
        showNotification(`Processing ${data.filename}...`);
        // Clear previous slides
        slides = [];
        totalSlides = 0;
        currentSlideNumber = 0;

        // Hide current slide and show loading message
        document.getElementById('noSlideMessage').style.display = 'block';
        document.getElementById('currentSlide').classList.add('hidden');
    });

    socket.on('total-slides', (data) => {
        totalSlides = data.totalSlides;
        slides = new Array(totalSlides).fill(null);  // Pre-allocate array with nulls
        console.log(`📊 Expecting ${totalSlides} slides`);
        showNotification(`Loading ${totalSlides} slides...`);
    });

    socket.on('slide-ready', (data) => {
        console.log("✅ New slide ready:", data);
        slides[data.index] = data.url;

        // Display first slide immediately when ready
        if (data.index === 0) {
            currentSlideNumber = 0;
            displaySlide(data.url);
            updateSlideInfo();
            showNotification('First slide ready!');
        }
    });

    socket.on('upload-complete', (data) => {
        console.log("✅ Upload complete:", data.totalSlides);
        showNotification(`All ${data.totalSlides} slides loaded successfully!`);

        // Ensure first slide is displayed if not already
        if (slides[0] && currentSlideNumber === 0) {
            displaySlide(slides[0]);
            updateSlideInfo();
        }
    });

    // Handle direct slide uploads (for compatibility)
    socket.on('slide-uploaded', (data) => {
        console.log('🔍 Frontend: Received slide-uploaded:', data);

        const slideData = data.slideData || [];
        slides = slideData.map(slide => slide.url || slide);
        totalSlides = slides.length;
        currentSlideNumber = 0;

        console.log('🔍 Frontend: Extracted slide URLs:', slides);

        if (slides.length > 0) {
            displaySlide(slides[currentSlideNumber]);
            updateSlideInfo();
            showNotification('Teacher uploaded new slides');
        }
    });

    socket.on('slide-changed', (data) => {
        currentSlideNumber = data.slideNumber;
        if (slides[currentSlideNumber]) {
            displaySlide(slides[currentSlideNumber]);
        }
        updateSlideInfo();
        showNotification(`Teacher changed to slide ${currentSlideNumber + 1}`);
```

```
});

socket.on('new-message', (message) => {
    displayMessage(message);
});

socket.on('teacher-left', () => {
    showNotification('Teacher has left the classroom', 'warning');
    // Stop audio if teacher leaves
    if (currentUser?.role === 'student' && isAudioStreaming) {
        updateAudioStatus('stopped', 'Audio: Teacher disconnected');
        peerConnections.forEach(pc => pc.close());
        peerConnections.clear();
    }
});

// WebRTC signaling handlers
socket.on('webrtc-offer', handleWebRTCOffer);
socket.on('webrtc-answer', handleWebRTCAnswer);
socket.on('webrtc-ice-candidate', handleWebRTCIceCandidate);

// Handle classroom state updates
socket.on('classroom-state', updateClassroomState);
socket.on('participants-updated', updateParticipantsList);

// Offline resources: when teacher adds a resource, ask SW to cache it
socket.on('resource-added', (resource) => {
    console.log('Resource announced:', resource);
    if (navigator.serviceWorker && navigator.serviceWorker.controller) {
        navigator.serviceWorker.controller.postMessage({
            type: 'CACHE_RESOURCE_URLS',
            payload: { urls: [resource.url] }
        });
    }
    addResourceToList(resource);
});

socket.on('resource-removed', (resource) => {
    console.log('Resource removed:', resource);
    if (navigator.serviceWorker && navigator.serviceWorker.controller) {
        navigator.serviceWorker.controller.postMessage({
            type: 'DELETE_RESOURCE_URLS',
            payload: { urls: [resource.url] }
        });
        }
        removeResourceFromList(resource);
    });
}

function loadSlides(slideData) {
    console.log('🔍 Frontend: Loading slides:', slideData);

    // Convert slide objects to URL array
    slides = slideData.map(slide => slide.url || slide);
    currentSlideNumber = 0;
    totalSlides = slides.length;

    console.log('🔍 Frontend: Processed slides:', slides);

    if(slides.length > 0) {
        document.getElementById('noSlideMessage').style.display = 'none';
        document.getElementById('currentSlide').classList.remove('hidden');
        showSlide(currentSlideNumber);
    } else {
        document.getElementById('noSlideMessage').style.display = 'block';
        document.getElementById('currentSlide').classList.add('hidden');
    }
}

// Resources: upload flow (teacher)
function triggerResourceUpload() {
    if (!currentUser || currentUser.role !== 'teacher') {
        alert('Only teachers can upload resources');
        return;
    }
    const input = document.getElementById('resourceUpload');
    if (input) input.click();
}

function handleResourceUpload(event) {
    const file = event.target.files && event.target.files[0];
    if (!file) return;

    const form = new FormData();
    form.append('file', file);

    fetch('/upload-resource', { method: 'POST', body: form })
        .then(r => {
            if (!r.ok) throw new Error('Upload failed');
            return r.json();
        })
        .then(data => {
            const res = data && data.resource;
            if (res) {
                // Add to list immediately; SW caching also triggered by socket echo
                addResourceToList(res);
                if (navigator.serviceWorker && navigator.serviceWorker.controller) {
                    navigator.serviceWorker.controller.postMessage({
                        type: 'CACHE_RESOURCE_URLS',
                        payload: { urls: [res.url] }
                    });
                }
            }
            // Reset input
            event.target.value = '';
        })
        .catch(err => {
            console.error('Resource upload failed', err);
            alert('Resource upload failed: ' + err.message);
        });
}

// Resources: initial index load and cache
function initResourcesIndex() {
    fetch('/resources-index')
        .then(r => r.json())
        .then(data => {
            const list = (data && data.resources) || [];
            // Render
            const container = document.getElementById('resourcesList');
            if (container) container.innerHTML = '';
            list.forEach(addResourceToList);
            // Ask SW to cache all
            const urls = list.map(x => x.url);
            if (urls.length && navigator.serviceWorker && navigator.serviceWorker.controller) {
                navigator.serviceWorker.controller.postMessage({
                    type: 'CACHE_RESOURCE_URLS',
                    payload: { urls }
                });
            }
        })
        .catch(() => {});
```

```
}

function addResourceToList(res) {
   const list = document.getElementById('resourcesList');
   if (!list || !res) return;
   const key = `${res.id || ''}:${res.name || res.safeName || res.url}`;
   // Prevent duplicates (e.g., immediate add after upload + socket echo)
   const alreadyExists = Array.from(list.children || []).some(n => n.dataset && n.dataset.key ===
key);
   if (alreadyExists) {
      // Ensure teacher controls exist on the existing row
      if (currentUser && currentUser.role === 'teacher') {
         ensureTeacherActionsOnResources();
      }
      return;
   }
   const row = document.createElement('div');
   row.style.display = 'flex';
   row.style.alignItems = 'center';
   row.style.justifyContent = 'space-between';
   row.style.gap = '8px';
   row.style.marginBottom = '6px';
   row.dataset.key = key;
   if (res.id) row.dataset.id = res.id;
   if (res.name || res.safeName) row.dataset.name = res.name || res.safeName;
   if (res.url) row.dataset.url = res.url;

   const nameEl = document.createElement('div');
   nameEl.style.flex = '1';
   nameEl.style.wordBreak = 'break-all';
   nameEl.textContent = res.name || res.safeName || res.url;

   const actions = document.createElement('div');
   actions.style.display = 'flex';
   actions.style.gap = '6px';
   actions.setAttribute('data-actions', 'true');

   const downloadBtn = document.createElement('a');
   downloadBtn.href = res.url;
   downloadBtn.textContent = 'Download';
   downloadBtn.setAttribute('download', res.safeName || '');
   downloadBtn.style.textDecoration = 'none';
   downloadBtn.style.padding = '6px 10px';
   downloadBtn.style.border = '1px solid #ddd';
```

```
    downloadBtn.style.borderRadius = '6px';
    downloadBtn.style.background = '#f8f8f8';

    actions.appendChild(downloadBtn);

    if (currentUser && currentUser.role === 'teacher') {
        appendRemoveButton(actions, res);
    }

    row.appendChild(nameEl);
    row.appendChild(actions);
    list.appendChild(row);
}

function removeResourceFromList(res) {
    const list = document.getElementById('resourcesList');
    if (!list || !res) return;
    const key = `${res.id || ''}:${res.name || res.safeName || res.url}`;
    const nodes = Array.from(list.children);
    for (const n of nodes) {
        if (n.dataset && n.dataset.key === key) {
            n.remove();
            break;
        }
    }
}

function deleteResource(res) {
    if (!res || !res.id || !res.name) {
        // Try to extract id/name from URL: /resources/:id/:name
        try {
            const parts = (res.url || '').split('/').filter(Boolean);
            const idx = parts.indexOf('resources');
            res.id = res.id || parts[idx + 1];
            res.name = res.name || parts[idx + 2];
        } catch {}
    }
    if (!res.id || !res.name) return alert('Invalid resource');

    fetch(`/resources/${encodeURIComponent(res.id)}/${encodeURIComponent(res.name)}`, {
method: 'DELETE' })
        .then(r => {
            if (!r.ok) throw new Error('Delete failed');
            removeResourceFromList(res);
            if (navigator.serviceWorker && navigator.serviceWorker.controller) {
                navigator.serviceWorker.controller.postMessage({
                    type: 'DELETE_RESOURCE_URLS',
                    payload: { urls: [res.url] }
                });
            }
        })
        .catch(err => {
            console.error('Failed to delete resource', err);
            alert('Failed to delete resource: ' + err.message);
        });
}

function appendRemoveButton(actionsEl, res) {
    const removeBtn = document.createElement('button');
    removeBtn.textContent = 'Remove';
    removeBtn.style.padding = '6px 10px';
    removeBtn.style.border = 'none';
    removeBtn.style.borderRadius = '6px';
    removeBtn.style.background = '#ff6b6b';
    removeBtn.style.color = '#fff';
    removeBtn.onclick = () => deleteResource(res);
    actionsEl.appendChild(removeBtn);
}

function ensureTeacherActionsOnResources() {
    try {
        if (!(currentUser && currentUser.role === 'teacher')) return;
        const list = document.getElementById('resourcesList');
        if (!list) return;
        const rows = Array.from(list.children || []);
        rows.forEach(row => {
            const actions = row.querySelector('[data-actions="true"]');
            if (!actions) return;
            const hasRemove = Array.from(actions.children).some(el => el.tagName ===
'BUTTON');
            if (!hasRemove) {
                const res = {
                    id: row.dataset.id,
                    name: row.dataset.name,
                    url: row.dataset.url
                };
                appendRemoveButton(actions, res);
            }
        });
    } catch (_) {}
}

function showSlide(slideIndex) {
    if (slides[slideIndex]) {
        displaySlide(slides[slideIndex]);
        updateSlideInfo();

        // Preload next slide for better performance
        if (slides[slideIndex + 1]) {
            const preload = new Image();
            preload.src = slides[slideIndex + 1];
        }
    }
}

function joinClassroom() {
    const name = document.getElementById('nameInput').value.trim();
    const role = document.getElementById('roleSelect').value;

    if (!name) {
        alert('Please enter your name');
        return;
    }

    currentUser = { name, role };

    // Send join request to server
    socket.emit('join-classroom', { name, role });

    // Show classroom interface
    document.getElementById('joinForm').style.display = 'none';
    document.getElementById('classroom').style.display = 'grid';

    // Show teacher controls if user is teacher
    if (role === 'teacher') {
        document.getElementById('teacherControls').style.display = 'flex';
        // Ensure existing resources display remove buttons
        ensureTeacherActionsOnResources();
    }

    console.log(`Joined as ${role}: ${name}`);
}
```

```
function updateStatus(status, text) {
    const statusEl = document.getElementById('status');
    if (statusEl) {
        statusEl.className = `status ${status}`;
        statusEl.textContent = text;
    }
}

function updateClassroomState(state) {
    console.log('🔍 Frontend: Received classroom state:', state);

    currentSlideNumber = state.currentSlide || 0;
    totalSlides = state.totalSlides || 0;

    // Handle slideData properly
    if (state.slideData && Array.isArray(state.slideData)) {
        slides = state.slideData.map(slide => slide.url || slide);
        console.log('🔍 Frontend: State slides:', slides);
    }

    if (slides.length > 0 && slides[currentSlideNumber]) {
        displaySlide(slides[currentSlideNumber]);
    }

    updateSlideInfo();
    if (state.participants) {
        updateParticipantsList(state.participants);
    }
}

function updateParticipantsList(participants) {
    const listEl = document.getElementById('participantsList');
    const countEl = document.getElementById('participantCount');

    if (listEl) {
        listEl.innerHTML = '';
    }
    if (countEl) {
        countEl.textContent = participants.length;
    }

    participants.forEach(participant => {
        const div = document.createElement('div');
        div.className = `participant ${participant.role}`;
        div.textContent = `${participant.role === 'teacher' ? '👨‍🏫' : '👨‍🎓'} ${participant.name}`;
        if (listEl) {
            listEl.appendChild(div);
        }
    });
}

function updateSlideInfo() {
    const slideInfoEl = document.getElementById('slideInfo');
    if (slideInfoEl) {
        slideInfoEl.textContent = `Slide ${currentSlideNumber + 1} of ${totalSlides || 1}`;
    }
}

function displaySlide(slideUrl) {
    console.log('🖼️ Frontend: Attempting to display slide:', slideUrl);

    const slideImg = document.getElementById('currentSlide');
    const noSlideMsg = document.getElementById('noSlideMessage');

    if (!slideUrl) {
        console.log('❌ Frontend: No slide URL provided');
        if (slideImg) slideImg.classList.add('hidden');
        if (noSlideMsg) noSlideMsg.style.display = 'block';
        return;
    }

    if (slideImg) {
        console.log('🔍 Frontend: Setting image src to:', slideUrl);

        // Add load event listener
        slideImg.onload = function() {
            console.log('✅ Frontend: Image loaded successfully:', slideUrl);

            // Preload next slide in the background
            if (slides && slides[currentSlideNumber + 1]) {
                const nextUrl = slides[currentSlideNumber + 1];
                console.log('📥 Preloading next slide:', nextUrl);
                const preload = new Image();
                preload.src = nextUrl;
            }
        };
        // Enhanced error handling
        slideImg.onerror = function(event) {
            console.error('❌ Frontend: Failed to load slide:', slideUrl);
            console.error('❌ Frontend: Error event:', event);

            // Try to fetch the URL to see what the actual error is
            fetch(slideUrl)
                .then(response => {
                    console.log('🔍 Frontend: Fetch test status:', response.status);
                    if (!response.ok) {
                        console.error('❌ Frontend: Server returned:', response.status,
response.statusText);
                    }
                })
                .catch(fetchError => {
                    console.error('❌ Frontend: Fetch error:', fetchError);
                });

            this.classList.add('hidden');
            if (noSlideMsg) noSlideMsg.style.display = 'block';
        };

        slideImg.src = slideUrl;
        slideImg.classList.remove('hidden');
    }

    if (noSlideMsg) noSlideMsg.style.display = 'none';
}

function nextSlide() {
    if (currentUser?.role === 'teacher' && currentSlideNumber < totalSlides - 1) {
        currentSlideNumber++;
        if (slides[currentSlideNumber]) {
            displaySlide(slides[currentSlideNumber]);
        }
        updateSlideInfo();

        // Notify students
        socket.emit('change-slide', { slideNumber: currentSlideNumber });
    }
}

function previousSlide() {
    if (currentUser?.role === 'teacher' && currentSlideNumber > 0) {
        currentSlideNumber--;
        if (slides[currentSlideNumber]) {
            displaySlide(slides[currentSlideNumber]);
        }
        updateSlideInfo();

        // Notify students
        socket.emit('change-slide', { slideNumber: currentSlideNumber });
    }
}

function triggerFileUpload() {
    if (!currentUser) {
        alert('Please join the classroom first');
        return;
    }

    if (currentUser.role !== 'teacher') {
        alert('Only teachers can upload slides');
        return;
    }

    const fileInput = document.getElementById('slideUpload');
    if (fileInput) {
        fileInput.click();
    } else {
        console.error('File input element not found');
    }
}

function handleFileUpload(event) {
    const file = event.target.files[0];
    if (!file || currentUser?.role !== 'teacher') return;

    console.log('📤 Frontend: Uploading file:', file.name);
    showNotification('Uploading slides...', 'info');

    const formData = new FormData();
    formData.append("file", file);

    fetch("/upload", { method: "POST", body: formData })
        .then(res => {
            console.log('📤 Frontend: Upload response status:', res.status);
            if (!res.ok) {
                throw new Error(`HTTP error! status: ${res.status}`);
            }
            return res.json();
        })
        .then(data => {
            console.log('📤 Frontend: Upload response data:', data);

            if (data.slides && Array.isArray(data.slides)) {
                // Convert slide objects to URLs
                slides = data.slides.map(slide => slide.url || slide);
                currentSlideNumber = 0;
                totalSlides = slides.length;

                console.log('📤 Frontend: Processed uploaded slides:', slides);

                if (slides.length > 0) {
                    displaySlide(slides[0]);
                    updateSlideInfo();
                    showNotification('Slides uploaded successfully!');
                } else {
                    throw new Error('No slides in response');
                }
            } else {
                console.error('❌ Frontend: Invalid response format:', data);
                throw new Error('Invalid response format');
            }
        })
        .catch(err => {
            console.error("❌ Frontend: Upload failed:", err);
            alert("Upload failed: " + err.message);
        });
}

// Chat functionality
function sendMessage() {
    const chatInput = document.getElementById('chatInput');
    if (!chatInput) return;

    const message = chatInput.value.trim();

    if (message && socket && currentUser) {
        socket.emit('send-message', {
            text: message,
            sender: currentUser.name,
            role: currentUser.role
        });
        chatInput.value = '';
    }
}

function setupChatEnterKey() {
    const chatInput = document.getElementById('chatInput');
    if (chatInput) {
        chatInput.addEventListener('keypress', function(e) {
            if (e.key === 'Enter') {
                sendMessage();
            }
        });
    }
}

function displayMessage(message) {
    const messagesEl = document.getElementById('chatMessages');
    if (!messagesEl) return;

    const messageDiv = document.createElement('div');
    messageDiv.className = `message ${message.role}`;

    messageDiv.innerHTML = `
        <div class="message-header">${message.sender} (${message.role})</div>
        <div class="message-content">${escapeHtml(message.text)}</div>
    `;

    messagesEl.appendChild(messageDiv);
    messagesEl.scrollTop = messagesEl.scrollHeight;
}

function escapeHtml(text) {
    const div = document.createElement('div');
    div.textContent = text;
    return div.innerHTML;
}

function showNotification(text, type = 'info') {
    const notification = document.createElement('div');
    notification.style.cssText = `
        position: fixed;
        top: 70px;
        right: 20px;
        padding: 10px 15px;
        border-radius: 6px;
        color: white;
        font-size: 14px;
        z-index: 1000;
        background: ${type === 'warning' ? '#ff9800' : type === 'error' ? '#f44336' : '#4CAF50'};
        box-shadow: 0 2px 8px rgba(0,0,0,0.2);
    `;
    notification.textContent = text;

    document.body.appendChild(notification);

    setTimeout(() => {
        if (notification.parentNode) {
            notification.remove();
        }
    }, 3000);
}

function showConnectedMessage() {
    const messages = [
        "🎉 Connected! You can now join the classroom.",
        "✨ Connection established successfully!",
        "🚀 Ready to start learning!"
    ];

    const randomMessage = messages[Math.floor(Math.random() * messages.length)];
    showNotification(randomMessage);
}

//
============================================================================
=====
// WebRTC Audio Streaming (Low-bandwidth with Opus codec)
//
============================================================================
=====

async function startAudio() {
    if (currentUser?.role !== 'teacher') {
        alert('Only teacher can start audio streaming');
        return;
    }
    if (isAudioStreaming) {
        console.log('Audio already streaming');
        return;
    }

    try {
        // Get user media with optimized audio settings
        localStream = await navigator.mediaDevices.getUserMedia(audioConstraints);

        console.log('Got local audio stream');

        // Update UI
        const startBtn = document.getElementById('startAudioBtn');
        const stopBtn = document.getElementById('stopAudioBtn');
        if (startBtn) startBtn.classList.add('hidden');
        if (stopBtn) stopBtn.classList.remove('hidden');

        updateAudioStatus('streaming', 'Audio: Streaming 🔴');

        isAudioStreaming = true;

        // Create peer connections for all current students
        await createPeerConnectionsForStudents();

    } catch (error) {
        console.error('Error accessing microphone:', error);
        alert('Could not access microphone. Please check permissions.');
        updateAudioStatus('error', 'Audio: Error accessing microphone');
    }
}

function stopAudio() {
    if (localStream) {
        localStream.getTracks().forEach(track => track.stop());
        localStream = null;
    }

    // Close all peer connections
    peerConnections.forEach(pc => {
        pc.close();
    });
    peerConnections.clear();
    // Update UI
    const startBtn = document.getElementById('startAudioBtn');
    const stopBtn = document.getElementById('stopAudioBtn');
    if (startBtn) startBtn.classList.remove('hidden');
    if (stopBtn) stopBtn.classList.add('hidden');

    updateAudioStatus('stopped', 'Audio: Stopped');

    isAudioStreaming = false;

    // Notify students that audio has stopped
    socket.emit('audio-stopped');

    console.log('Audio streaming stopped');
}

function updateAudioStatus(status, text) {
    const statusEl = document.getElementById('audioStatus');
    if (statusEl) {
        statusEl.className = `audio-status ${status}`;
        statusEl.textContent = text;
    }
}

async function createPeerConnectionsForStudents() {
    if (!localStream) {
        console.error('No local stream available');
        return;
    }

    try {
        // Create a single peer connection for broadcasting
        const peerConnection = new RTCPeerConnection(rtcConfiguration);

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

        // Handle ICE candidates
        peerConnection.onicecandidate = (event) => {
            if (event.candidate) {
                socket.emit('webrtc-ice-candidate', {
                    candidate: event.candidate
                });
            }
        };

        // Handle connection state changes
        peerConnection.onconnectionstatechange = () => {
            console.log('Connection state:', peerConnection.connectionState);
            if (peerConnection.connectionState === 'failed') {
                console.error('WebRTC connection failed');
                updateAudioStatus('error', 'Audio: Connection failed');
            }
        };

        // Create and send offer
        const offer = await peerConnection.createOffer();
        await peerConnection.setLocalDescription(offer);

        socket.emit('webrtc-offer', { offer: offer });

        // Store the peer connection
        peerConnections.set('broadcast', peerConnection);

        console.log('Created WebRTC offer for audio streaming');

    } catch (error) {
        console.error('Error creating peer connection:', error);
        updateAudioStatus('error', 'Audio: Failed to create connection');
    }
}

// Handle incoming WebRTC offer (students receive this)
async function handleWebRTCOffer(data) {
    if (currentUser?.role !== 'student') return;

    console.log('Received WebRTC offer from teacher');

    try {
        const peerConnection = new RTCPeerConnection(rtcConfiguration);

        // Handle incoming audio stream
        peerConnection.ontrack = (event) => {
            console.log('Received remote audio stream');
            const remoteAudio = new Audio();
            remoteAudio.srcObject = event.streams[0];
            remoteAudio.autoplay = true;

            // Handle audio play promise
            remoteAudio.play().catch(error => {
                console.error('Error playing audio:', error);
                updateAudioStatus('error', 'Audio: Playback error');
            });

            updateAudioStatus('receiving', 'Audio: Receiving from teacher 🔊');
        };

        // Handle ICE candidates
        peerConnection.onicecandidate = (event) => {
            if (event.candidate) {
                socket.emit('webrtc-ice-candidate', {
                    candidate: event.candidate,
                    targetId: data.senderId
                });
            }
        };

        // Handle connection state changes
        peerConnection.onconnectionstatechange = () => {
            console.log('Student connection state:', peerConnection.connectionState);
            if (peerConnection.connectionState === 'disconnected' ||
                peerConnection.connectionState === 'failed') {
                updateAudioStatus('stopped', 'Audio: Connection lost');
            }
        };

        // Set remote description and create answer
        await peerConnection.setRemoteDescription(data.offer);
        const answer = await peerConnection.createAnswer();
        await peerConnection.setLocalDescription(answer);

        // Send answer back to teacher
        socket.emit('webrtc-answer', {
            answer: answer,
            targetId: data.senderId
        });

        // Store the peer connection
        peerConnections.set(data.senderId || 'teacher', peerConnection);
```

```
    } catch (error) {
        console.error('Error handling WebRTC offer:', error);
        updateAudioStatus('error', 'Audio: Connection error');
    }
}

// Handle WebRTC answer (teacher receives this)
async function handleWebRTCAnswer(data) {
    if (currentUser?.role !== 'teacher') return;

    console.log('Received WebRTC answer from student');

    try {
        const peerConnection = peerConnections.get('broadcast');
        if (peerConnection && peerConnection.signalingState !== 'stable') {
            await peerConnection.setRemoteDescription(data.answer);
        }
    } catch (error) {
        console.error('Error handling WebRTC answer:', error);
    }
}

// Handle ICE candidates
async function handleWebRTCIceCandidate(data) {
    console.log('Received ICE candidate');

    if (!data.candidate) return;

    try {
        let peerConnection;
        if (currentUser?.role === 'teacher') {
            peerConnection = peerConnections.get('broadcast');
        } else {
            peerConnection = peerConnections.get(data.senderId || 'teacher');
        }

        if (peerConnection && peerConnection.remoteDescription) {
            await peerConnection.addIceCandidate(data.candidate);
        } else {
            console.log('Peer connection not ready for ICE candidate');
        }
    } catch (error) {
        console.error('Error adding ICE candidate:', error);
    }
```

```
}

//
=========================================================================
=====
// Debugging and Utility Functions
//
=========================================================================
=====

function debugSlideState() {
    console.log('=== SLIDE DEBUG INFO ===');
    console.log('Current slide number:', currentSlideNumber);
    console.log('Total slides:', totalSlides);
    console.log('Slides array:', slides);
    console.log('Current user:', currentUser);

    // Check DOM elements
    const slideImg = document.getElementById('currentSlide');
    const noSlideMsg = document.getElementById('noSlideMessage');

    console.log('Slide image element:', slideImg);
    console.log('No slide message element:', noSlideMsg);

    if (slideImg) {
        console.log('Slide image src:', slideImg.src);
        console.log('Slide image classes:', slideImg.className);
    }

    console.log('========================');
}

// Add this to window for debugging in console
window.debugSlideState = debugSlideState;

// Handle page visibility changes to manage connections
document.addEventListener('visibilitychange', function() {
    if (document.hidden && isAudioStreaming && currentUser?.role === 'teacher') {
        console.log('Page hidden, maintaining audio connection');
    } else if (!document.hidden && isAudioStreaming) {
        console.log('Page visible, audio connection active');
    }
});
```

```
// Clean up on page unload
window.addEventListener('beforeunload', function() {
    if (localStream) {
        localStream.getTracks().forEach(track => track.stop());
    }
    peerConnections.forEach(pc => pc.close());
    if (socket) {
        socket.disconnect();
    }
});
</script>
</body>
</html>
```

Public\ Sw.js

```
const CACHE_NAME = 'luminex-cache-v1';
const RESOURCE_CACHE = 'luminex-resources-v1';

self.addEventListener('install', (event) => {
  event.waitUntil((async () => {
    const cache = await caches.open(CACHE_NAME);
    await cache.addAll([
      '/',
      '/index.html'
    ]);
    self.skipWaiting();
  })());
});

self.addEventListener('activate', (event) => {
  event.waitUntil((async () => {
    const keys = await caches.keys();
    await Promise.all(
      keys.filter(k => ![CACHE_NAME, RESOURCE_CACHE].includes(k)).map(k =>
caches.delete(k))
    );
    self.clients.claim();
  })());
});

// Cache-first for resources; network-first for others
self.addEventListener('fetch', (event) => {
  const url = new URL(event.request.url);
  if (url.pathname.startsWith('/resources/')) {
    event.respondWith(cacheFirst(event.request));
    return;
  }

  // Default: try network then fallback to cache
  event.respondWith(networkThenCache(event.request));
});

async function cacheFirst(request) {
  const cache = await caches.open(RESOURCE_CACHE);
  const cached = await cache.match(request, { ignoreVary: true });
  if (cached) return cached;
  try {
    const response = await fetch(request);
    if (response && response.ok) {
      cache.put(request, response.clone());
    }
    return response;
  } catch (e) {
    return cached || Response.error();
  }
}

async function networkThenCache(request) {
  try {
    const response = await fetch(request);
    const cache = await caches.open(CACHE_NAME);
    if (response && response.ok && request.method === 'GET') {
      cache.put(request, response.clone());
    }
    return response;
  } catch (e) {
    const cache = await caches.open(CACHE_NAME);
    const cached = await cache.match(request, { ignoreVary: true });
    return cached || Response.error();
  }
}

// Message API to pre-cache resource URLs
self.addEventListener('message', async (event) => {
  const { type, payload } = event.data || {};
  if (type === 'CACHE_RESOURCE_URLS' && payload && Array.isArray(payload.urls)) {
    const cache = await caches.open(RESOURCE_CACHE);
    await Promise.all(payload.urls.map(async (u) => {
      try {
        const req = new Request(u, { mode: 'same-origin' });
        const resp = await fetch(req);
        if (resp && resp.ok) {
          await cache.put(req, resp.clone());
        }
      } catch (_) {}
    }));
    return;
  }
  if (type === 'DELETE_RESOURCE_URLS' && payload && Array.isArray(payload.urls)) {
    const cache = await caches.open(RESOURCE_CACHE);
    await Promise.all(payload.urls.map(async (u) => {
      try {
        const req = new Request(u, { mode: 'same-origin' });
        await cache.delete(req, { ignoreVary: true });
      } catch (_) {}
    }));
    return;
  }
});
```

server.js

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');
const path = require('path');
const multer = require("multer");
const { v4: uuidv4 } = require("uuid");
const fs = require("fs");
const sharp = require("sharp");
const pdfPoppler = require("pdf-poppler");
const { exec } = require("child_process");

const app = express();
const server = http.createServer(app);
const io = socketIo(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"]
  }
});
```

```
// });
```

```
// Add this line after your other middleware (before your routes)
app.use('/slides', express.static(path.join(__dirname, 'slides')));
// Serve downloadable resources
app.use('/resources', express.static(path.join(__dirname, 'resources')));

// app.use('/slides', (req, res, next) => {
//   console.log('🔍 Slide request:', req.url);

//   // Convert URL path to proper file system path
//   const requestPath = req.url.replace(/\//g, path.sep);
//   const fullPath = path.join(__dirname, 'slides', requestPath);

//   console.log('🔍 Converted path:', fullPath);
//   console.log('🔍 File exists:', fs.existsSync(fullPath));

//   // If file exists, serve it manually to avoid path issues
//   if (fs.existsSync(fullPath)) {
//     console.log('✅ Serving file manually:', fullPath);

//     // Set proper headers
//     res.setHeader('Content-Type', 'image/jpeg');
//     res.setHeader('Cache-Control', 'public, max-age=3600');

//     // Read and send file
//     const fileStream = fs.createReadStream(fullPath);
//     fileStream.pipe(res);

//     fileStream.on('error', (err) => {
//       console.error('❌ Error reading file:', err);
//       res.status(500).send('Error reading file');
//     });

//     fileStream.on('end', () => {
//       console.log('✅ File sent successfully');
//     });

//     return; // Don't call next()
//   }

//   // If file doesn't exist, log details and continue to static middleware
//   console.log('❌ File not found, trying static middleware');
//   next();
// });
// // Keep the static middleware as backup

// app.use(express.static(path.join(__dirname, 'public')));

// Add this route - it will handle /slides/:id/:filename manually
// REPLACE your current slides routing with this:

// 1. REMOVE all the app.use('/slides', ...) middleware

// 2. KEEP ONLY this specific route:
app.get('/slides/:id/:filename', (req, res) => {
  console.log('🎯 MANUAL ROUTE CALLED for:', req.params);

  const { id, filename } = req.params;
  const filePath = path.join(__dirname, 'slides', id, filename);

  console.log('🎯 Looking for file:', filePath);
  console.log('🎯 File exists:', fs.existsSync(filePath));

  if (fs.existsSync(filePath)) {
    console.log('✅ File found, sending...');

    // Set proper headers
    const ext = path.extname(filename).toLowerCase();
    if (ext === '.jpg' || ext === '.jpeg') {
      res.setHeader('Content-Type', 'image/jpeg');
    } else if (ext === '.png') {
      res.setHeader('Content-Type', 'image/png');
    }

    res.setHeader('Cache-Control', 'public, max-age=3600');

    // Send file using absolute path
    res.sendFile(path.resolve(filePath), (err) => {
      if (err) {
        console.error('❌ Error sending file:', err);
        res.status(500).send('Error sending file');
      } else {
        console.log('✅ File sent successfully!');
      }
    });

  } else {
    console.log('❌ File not found');

    // Debug: show what files exist in that directory
    const dirPath = path.join(__dirname, 'slides', id);
    if (fs.existsSync(dirPath)) {
      const files = fs.readdirSync(dirPath);
      console.log('📂 Available files in directory:', files);

      res.status(404).json({
        error: 'File not found',
        requested: filename,
        directory: id,
        availableFiles: files
      });
    } else {
      console.log('📂 Directory does not exist:', dirPath);

      // Show all available slide directories
      const slidesDir = path.join(__dirname, 'slides');
      const availableDirs = fs.existsSync(slidesDir) ? fs.readdirSync(slidesDir) : [];

      res.status(404).json({
        error: 'Directory not found',
        requested: id,
        availableDirectories: availableDirs
      });
    }
  }
});

// 3. KEEP your public static files
app.use(express.static(path.join(__dirname, 'public')));
app.use('/slides', express.static(path.join(__dirname, 'slides'), {
  setHeaders: (res, filePath) => {
    console.log('📤 Static middleware serving:', filePath);
  }
}));

// 4. ADD a test route to verify everything works
app.get('/test-slides', (req, res) => {
  const slidesDir = path.join(__dirname, 'slides');

  if (!fs.existsSync(slidesDir)) {
    return res.json({ error: 'Slides directory does not exist' });
  }

  try {
    const directories = fs.readdirSync(slidesDir).filter(item => {
      return fs.statSync(path.join(slidesDir, item)).isDirectory();
    });

    const result = {};
    directories.forEach(dir => {
      const dirPath = path.join(slidesDir, dir);
      result[dir] = fs.readdirSync(dirPath);
    });

    res.json({
      success: true,
      slidesDirectory: slidesDir,
      slideDirectories: result,
      totalDirectories: directories.length
    });

  } catch (error) {
    res.json({
      error: error.message,
      slidesDirectory: slidesDir
    });
  }
});
// Make sure you don't have duplicate /slides routes!

// Store classroom state
let classroomState = {
  currentSlide: 0,
  totalSlides: 0,
  slideData: [], // Array of slide URLs
  isTeacherPresent: false,
  participants: [],
  preloadedSlides: new Set(), // Track which slides are ready
  preloadQueue: [], // Queue of slides being processed
  preloadBuffer: 3, // How many slides ahead to preload
  isPreloading: false
};

// Store connected clients
let connectedClients = new Map();
```

```
// Enhanced multer configuration with limits
const upload = multer({
  dest: "uploads/",
  limits: {
    fileSize: 50 * 1024 * 1024 // 50MB limit
  },
  fileFilter: (req, file, cb) => {
    const allowedTypes = ['.pdf', '.pptx', '.png', '.jpg', '.jpeg'];
    const ext = path.extname(file.originalname).toLowerCase();

    if (allowedTypes.includes(ext)) {
      cb(null, true);
    } else {
      cb(new Error('Unsupported file type'), false);
    }
  }
});

// Upload and publish downloadable resources (any file)
app.post('/upload-resource', upload.single('file'), async (req, res) => {
  try {
    const file = req.file;
    if (!file) {
      return res.status(400).json({ error: 'No file uploaded' });
    }

    const resourceId = uuidv4();
    const resourcesBase = path.join(__dirname, 'resources', resourceId);
    if (!fs.existsSync(resourcesBase)) {
      fs.mkdirSync(resourcesBase, { recursive: true });
    }

    const originalName = file.originalname;
    const safeName = sanitizeFilename(originalName);
    const targetPath = path.join(resourcesBase, safeName);

    // Move from uploads temp to resources
    fs.renameSync(file.path, targetPath);

    const url = `/resources/${resourceId}/${safeName}`;
    const stats = fs.statSync(targetPath);

    // Notify all connected clients (students) to cache this resource
    const payload = {
      id: resourceId,
      name: originalName,
      safeName,
      url,
      size: stats.size,
      mime: req.headers['content-type'] || 'application/octet-stream',
      timestamp: Date.now()
    };
    io.emit('resource-added', payload);

    return res.json({ success: true, resource: payload });
  } catch (error) {
    console.error('Upload resource error:', error);
    return res.status(500).json({ error: error.message || 'Failed to upload resource' });
  } finally {
    if (req.file && fs.existsSync(req.file.path)) {
      try { fs.unlinkSync(req.file.path); } catch (_) {}
    }
  }
});

// Delete a resource (teacher action)
app.delete('/resources/:id/:name', (req, res) => {
  try {
    const { id, name } = req.params;
    const dir = path.join(__dirname, 'resources', id);
    const filePath = path.join(dir, name);

    if (!fs.existsSync(filePath)) {
      return res.status(404).json({ error: 'Resource not found' });
    }

    fs.unlinkSync(filePath);

    // If directory is now empty, remove it
    try {
      const remaining = fs.readdirSync(dir);
      if (remaining.length === 0) fs.rmdirSync(dir);
    } catch {}

    const url = `/resources/${id}/${name}`;
    io.emit('resource-removed', { id, name, url, timestamp: Date.now() });
    return res.json({ success: true });
  } catch (e) {
    console.error('Delete resource error:', e);
    return res.status(500).json({ error: e.message || 'Failed to delete resource' });
  }
});

// List available resources for initial load
app.get('/resources-index', (req, res) => {
  try {
    const resourcesDir = path.join(__dirname, 'resources');
    if (!fs.existsSync(resourcesDir)) return res.json({ resources: [] });

    const dirs = fs.readdirSync(resourcesDir).filter(name => {
      try { return fs.statSync(path.join(resourcesDir, name)).isDirectory(); } catch { return false; }
    });

    const resources = [];
    dirs.forEach(dir => {
      const dirPath = path.join(resourcesDir, dir);
      const files = fs.readdirSync(dirPath);
      files.forEach(file => {
        const filePath = path.join(dirPath, file);
        try {
          const stat = fs.statSync(filePath);
          resources.push({
            id: dir,
            name: file,
            url: `/resources/${dir}/${file}`,
            size: stat.size,
            mtimeMs: stat.mtimeMs
          });
        } catch {}
      });
    });

    // Sort by modified time desc
    resources.sort((a, b) => b.mtimeMs - a.mtimeMs);
    res.json({ resources });
  } catch (e) {
    res.status(500).json({ error: e.message || 'Failed to read resources' });
  }
});
```

```
async function autoPreloadSlides(currentSlide, classroomId, io) {
  if (classroomState.isPreloading) {
    console.log("⏳ Already preloading, skipping...");
    return;
  }

  classroomState.isPreloading = true;

  try {
    const slidesToPreload = [];

    // Determine which slides need preloading
    for (let i = 1; i <= classroomState.preloadBuffer; i++) {
      const nextSlideIndex = currentSlide + i;

      if (nextSlideIndex < classroomState.totalSlides &&
          !classroomState.preloadedSlides.has(nextSlideIndex)) {

        slidesToPreload.push(nextSlideIndex);
      }
    }

    if (slidesToPreload.length === 0) {
      console.log("✅ All nearby slides already preloaded");
      classroomState.isPreloading = false;
      return;
    }

    console.log(`🚀 Auto-preloading slides: ${slidesToPreload.map(s => s + 1).join(', ')}`);

    // Emit preload start notification
    io.emit("preload-started", {
      classroomId,
      slidesToPreload,
      currentSlide,
      timestamp: Date.now()
    });

    // Process slides in parallel but with controlled concurrency
    await processSlidePreloads(slidesToPreload, classroomId, io);

    console.log("✅ Auto-preloading completed");

  } catch (error) {
    console.error("❌ Auto-preload failed:", error);
  } finally {
    classroomState.isPreloading = false;
  }
}

async function preloadSingleSlide(slideIndex, classroomId, io) {
  try {
    if (classroomState.preloadedSlides.has(slideIndex)) {
      return; // Already preloaded
    }

    const slideData = classroomState.slideData[slideIndex];
    if (!slideData) {
      console.warn(`⚠️ Slide ${slideIndex + 1} data not found`);
      return;
    }

    // Get the actual file path
    const slidePath = path.join(__dirname, 'slides', classroomId, slideData.name);

    if (!fs.existsSync(slidePath)) {
      console.warn(`⚠️ Slide file not found: ${slidePath}`);
      return;
    }

    // Mark as preloaded (the file is already processed and ready)
    classroomState.preloadedSlides.add(slideIndex);

    // Notify clients that slide is ready for instant loading
    io.emit("slide-preloaded", {
      classroomId,
      slideIndex,
      url: slideData.url,
      fileSize: fs.statSync(slidePath).size,
      timestamp: Date.now()
    });

    console.log(`✅ Slide ${slideIndex + 1} preloaded and ready`);

  } catch (error) {
    console.error(`❌ Failed to preload slide ${slideIndex + 1}:`, error);
  }
}
```

```
// Enhanced upload endpoint with better error handling
app.post("/upload", upload.single("file"), async (req, res) => {
  try {
    const file = req.file;
    if (!file) {
      return res.status(400).json({ error: "No file uploaded" });
    }

    const ext = path.extname(file.originalname).toLowerCase();
    const id = uuidv4();
    const outDir = path.join(__dirname, "slides", id);

    // Emit upload started
    io.emit("upload-started", {
      classroomId: id,
      filename: file.originalname,
      timestamp: Date.now()
    });

    // Create output directory
    if (!fs.existsSync(outDir)) {
      fs.mkdirSync(outDir, { recursive: true });
    }

    let images = [];

    try {
      if (ext === ".pdf") {
        console.log("📄 Converting PDF...");
        images = await convertPdfToImages(file.path, outDir, io, id);
      } else if (ext === ".pptx") {
        console.log("📊 Converting PPTX to PDF...");
        const pdfPath = file.path + ".pdf";
        await convertPptToPdf(file.path, pdfPath);
        images = await convertPdfToImages(pdfPath, outDir, io, id);

        // Clean up temporary PDF
        if (fs.existsSync(pdfPath)) {
          fs.unlinkSync(pdfPath);
        }
      } else if ([".png", ".jpg", ".jpeg"].includes(ext)) {
        console.log("🖼️ Processing image...");
        const outPath = path.join(outDir, `slide-1.jpg`);

        // Process and compress image
        await sharp(file.path)
          .resize(1280, null, { withoutEnlargement: true })
          .jpeg({ quality: 60, mozjpeg: true, progressive: true })
          .toFile(outPath);

        // Emit total slides for single image
        io.emit("total-slides", {
          classroomId: id,
          totalSlides: 1
        });

        // Emit slide ready
        io.emit("slide-ready", {
          classroomId: id,
          url: `/slides/${id}/slide-1.jpg`,
          index: 0
        });

        // Add to images array
        images = [{
          url: `/slides/${id}/slide-1.jpg`,
          name: 'slide-1.jpg',
          index: 0
        }];
      } else {
        throw new Error(`Unsupported file type: ${ext}`);
      }

      if (images.length === 0) {
        throw new Error("No slides generated from file");
      }

      // Update classroom state
      classroomState.slideData = images;
      classroomState.totalSlides = images.length;
      classroomState.currentSlide = 0;

      console.log("✅ Generated images:", images.length);

      res.json({
        success: true,
        slides: images,
        totalSlides: images.length,
        classroomId: id
      });

    } catch (processingError) {
      console.error("File processing error:", processingError);

      // Clean up on error
      if (fs.existsSync(outDir)) {
        fs.rmSync(outDir, { recursive: true, force: true });
      }

      throw processingError;
    }

  } catch (err) {
    console.error("Upload error:", err);
    res.status(500).json({
      error: err.message || "File processing failed",
      details: process.env.NODE_ENV === 'development' ? err.stack : undefined
    });
  } finally {
    // Always clean up uploaded file
    if (req.file && fs.existsSync(req.file.path)) {
      fs.unlinkSync(req.file.path);
    }
  }
});

// Utility function to sanitize filenames
function sanitizeFilename(filename) {
  return filename.replace(/[^a-zA-Z0-9.-]/g, '_');
}

// Enhanced PDF to images conversion - NOW RETURNS IMAGE ARRAY
// async function convertPdfToImages(pdfPath, outDir, io, classroomId) {
//   try {
//     const opts = {
//       format: "png",
//       out_dir: outDir,
//       out_prefix: "page",
//       page: null,
//     };
//    // Convert PDF → PNGs
//    await pdfPoppler.convert(pdfPath, opts);

//    let files = fs.readdirSync(outDir)
//      .filter(f => f.toLowerCase().endsWith('.png'))
//      .sort((a, b) => {
//        const numA = parseInt(a.match(/\d+/)?.[0] || "0", 10);
//        const numB = parseInt(b.match(/\d+/)?.[0] || "0", 10);
//        return numA - numB;
//      });

//    if (!files.length) throw new Error(`No slides generated from ${pdfPath}`);

//    console.log(`📊 Total slides to process: ${files.length}`);
//    io.emit("total-slides", {
//      classroomId,
//      totalSlides: files.length
//    });

//    const images = []; // Array to collect processed slides

//    // Process slides one by one (so order is preserved)
//    for (let i = 0; i < files.length; i++) {
//      const file = files[i];
//      const filePath = path.join(outDir, file);
//      const outputFilename = `slide-${i + 1}.webp`;
//      const outPath = path.join(outDir, outputFilename);

//      await sharp(filePath)
//        .resize({ width: 720, withoutEnlargement: true })
//        .webp({ quality: 60 })
//        .toFile(outPath);

//      // Delete original PNG
//      fs.unlinkSync(filePath);

//      // Add to images array
//      const slideData = {
//        url: `/slides/${classroomId}/${outputFilename}`,
//        name: outputFilename,
//        index: i
//      };
//      images.push(slideData);
//      // Emit slide immediately after it's ready
//      io.emit("slide-ready", {
//        classroomId,
//        url: slideData.url,
//        index: i
//      });
//    }

//    console.log(`✅ PDF converted progressively in ${outDir}`);
//    return images; // Return the processed images array
//  } catch (err) {
//    console.error("PDF conversion failed:", err);
//    throw err;
//  }
// }


async function convertPdfToImages(pdfPath, outDir, io, classroomId) {
  try {
    const opts = {
      format: "png",
      out_dir: outDir,
      out_prefix: "page",
      page: null,
    };

    // Convert PDF → PNGs
    await pdfPoppler.convert(pdfPath, opts);

    let files = fs.readdirSync(outDir)
      .filter(f => f.toLowerCase().endsWith('.png'))
      .sort((a, b) => {
        const numA = parseInt(a.match(/\d+/)?.[0] || "0", 10);
        const numB = parseInt(b.match(/\d+/)?.[0] || "0", 10);
        return numA - numB;
      });

    if (!files.length) throw new Error(`No slides generated from ${pdfPath}`);

    console.log(`📊 Total slides to process: ${files.length}`);
    io.emit("total-slides", {
      classroomId,
      totalSlides: files.length
```

```
});

const images = [];
classroomState.preloadedSlides = new Set();

// Process slides one by one
for (let i = 0; i < files.length; i++) {
  const file = files[i];
  const filePath = path.join(outDir, file);
  const outputFilename = `slide-${i + 1}.webp`;
  const outPath = path.join(outDir, outputFilename);

  await sharp(filePath)
    .resize({ width: 720, withoutEnlargement: true })
    .webp({ quality: 60 })
    .toFile(outPath);

  // Delete original PNG
  fs.unlinkSync(filePath);

  const slideData = {
    url: `/slides/${classroomId}/${outputFilename}`,
    name: outputFilename,
    index: i
  };
  images.push(slideData);

  // Emit slide immediately after it's ready
  io.emit("slide-ready", {
    classroomId,
    url: slideData.url,
    index: i
  });

  // Preload first 4 slides
  if (i < 4) {
    classroomState.preloadedSlides.add(i);

    io.emit("slide-preloaded", {
      classroomId,
      slideIndex: i,
      url: slideData.url,
      timestamp: Date.now()
    });
```

```
      }
    }

    console.log(`✅ PDF converted progressively in ${outDir}`);
    return images;
  } catch (err) {
    console.error("PDF conversion failed:", err);
    throw err;
  }
}


function convertPptToPdf(inputPath, outputPath) {
  return new Promise((resolve, reject) => {
    const timeout = setTimeout(() => {
      reject(new Error('LibreOffice conversion timeout'));
    }, 30000); // 30 second timeout

    // Extract directory from outputPath for LibreOffice
    const outputDir = path.dirname(outputPath);

    // Ensure output directory exists
    if (!fs.existsSync(outputDir)) {
      fs.mkdirSync(outputDir, { recursive: true });
    }

    console.log(`Converting PPTX: ${inputPath} -> ${outputPath}`);

    const command = `soffice --headless --convert-to pdf --outdir "${outputDir}" "${inputPath}"`;
    console.log('Executing LibreOffice command:', command);

    exec(command, (err, stdout, stderr) => {
      clearTimeout(timeout);

      if (err) {
        console.error('LibreOffice error:', err);
        console.error('LibreOffice stderr:', stderr);
        reject(new Error(`PPTX conversion failed: ${err.message}`));
        return;
      }

      console.log('LibreOffice stdout:', stdout);

      // LibreOffice creates PDF with same base name as input file
      const inputBaseName = path.basename(inputPath, path.extname(inputPath));
      const generatedPdfPath = path.join(outputDir, inputBaseName + '.pdf');

      console.log('Looking for generated PDF at:', generatedPdfPath);

      // Wait a moment for file system to update
      setTimeout(() => {
        if (fs.existsSync(generatedPdfPath)) {
          // Move to desired output path if different
          if (generatedPdfPath !== outputPath) {
            try {
              fs.renameSync(generatedPdfPath, outputPath);
              console.log('✅ PDF successfully renamed to:', outputPath);
            } catch (renameErr) {
              console.error('Failed to rename PDF:', renameErr);
              reject(new Error(`Failed to rename PDF: ${renameErr.message}`));
              return;
            }
          }
          resolve();
        } else {
          // List all files in output directory for debugging
          console.error('Generated PDF not found. Files in output directory:');
          try {
            const files = fs.readdirSync(outputDir);
            console.error('Files:', files);
          } catch (listErr) {
            console.error('Could not list directory:', listErr);
          }
          reject(new Error(`PDF not generated at expected location: ${generatedPdfPath}`));
        }
      }, 1000); // Wait 1 second for file system
    });
  });
}

// Remove all slide directories immediately (used when teacher leaves)
function clearSlidesDirectory() {
  try {
    const slidesDir = path.join(__dirname, 'slides');
    if (!fs.existsSync(slidesDir)) return;
    const entries = fs.readdirSync(slidesDir);
    entries.forEach(entry => {
      const p = path.join(slidesDir, entry);
      try {
        const st = fs.statSync(p);
        if (st.isDirectory()) {
          fs.rmSync(p, { recursive: true, force: true });
        }
      } catch (e) {
        console.error('Failed to remove slide directory:', p, e.message);
      }
    });
  } catch (e) {
    console.error('clearSlidesDirectory error:', e.message);
  }
}

io.on('connection', (socket) => {
  console.log(`New client connected: ${socket.id}`);

  socket.on('join-classroom', (data) => {
    const { role, name } = data;

    if (!role || !name) {
      socket.emit('error', { message: 'Role and name are required' });
      return;
    }

    connectedClients.set(socket.id, { role, name, socketId: socket.id, joinedAt: Date.now() });

    if (role === 'teacher') classroomState.isTeacherPresent = true;

    // Update participants list
    classroomState.participants = Array.from(connectedClients.values());

    // Send current classroom state to the new client
    socket.emit('classroom-state', classroomState);

    // Notify all clients about updated participant list
    io.emit('participants-updated', classroomState.participants);

    console.log("User connected:", socket.id);

  // When teacher changes slide, trigger auto-preloading
  socket.on("teacher-slide-change", (data) => {
    const { classroomId, currentSlide, totalSlides } = data;
    // Update classroom state
    classroomState.currentSlide = currentSlide;

    // Broadcast to all clients
    socket.broadcast.emit("slide-changed", {
      classroomId,
      currentSlide,
      totalSlides,
      timestamp: Date.now()
    });

    // NEW: Trigger auto-preloading for upcoming slides
    setTimeout(() => {
      autoPreloadSlides(currentSlide, classroomId, io);
    }, 100); // Small delay to let slide change complete

    console.log(`📊 Changed to slide ${currentSlide + 1}/${totalSlides}`);
  });

  // NEW: Manual preload trigger (optional)
  socket.on("trigger-preload", (data) => {
    const { classroomId, currentSlide } = data;
    autoPreloadSlides(currentSlide, classroomId, io);
  });

  socket.on("disconnect", () => {
    console.log("User disconnected:", socket.id);
  });
  });

  // **Handle client disconnect**
  socket.on('disconnect', () => {
    console.log(`Client disconnected: ${socket.id}`);

    const client = connectedClients.get(socket.id);
    if (client) {
      connectedClients.delete(socket.id);

      // Update teacher presence if needed
      if (client.role === 'teacher') classroomState.isTeacherPresent = false;

      // Update participants list
      classroomState.participants = Array.from(connectedClients.values());
```

```
      // Notify all clients about updated participant list
      io.emit('participants-updated', classroomState.participants);

      // If the teacher left, clear generated slides immediately
      if (client.role === 'teacher') {
        try {
          // Reset slide state
          classroomState.slideData = [];
          classroomState.totalSlides = 0;
          classroomState.currentSlide = 0;
          classroomState.preloadedSlides = new Set();
          // Clear slide files on disk
          clearSlidesDirectory();
          // Optionally notify clients so UI can react if needed
          io.emit('slides-cleared', { timestamp: Date.now() });
        } catch (e) {
          console.error('Error clearing slides after teacher left:', e.message);
        }
      }
    }
  });


  // Handle slide changes (teacher only)
  socket.on('change-slide', (data) => {
    const client = connectedClients.get(socket.id);

    if (!client || client.role !== 'teacher') {
      socket.emit('error', { message: 'Only teachers can change slides' });
      return;
    }

    const slideNumber = parseInt(data.slideNumber);
    if (isNaN(slideNumber) || slideNumber < 0 || slideNumber >= classroomState.totalSlides) {
      socket.emit('error', { message: 'Invalid slide number' });
      return;
    }

    classroomState.currentSlide = slideNumber;

    // Send update to all clients
    io.emit('slide-changed', {
      slideNumber: slideNumber,
      timestamp: Date.now()
  });

  console.log(`Teacher changed to slide ${slideNumber}`);
});

// Handle chat messages
socket.on('send-message', (data) => {
  const client = connectedClients.get(socket.id);
  if (!client) {
    return;
  }

  if (!data.text || data.text.trim().length === 0) {
    socket.emit('error', { message: 'Message cannot be empty' });
    return;
  }

  // Limit message length
  const messageText = data.text.trim().substring(0, 500);

  const message = {
    id: Date.now(),
    sender: client.name,
    role: client.role,
    text: messageText,
    timestamp: Date.now()
  };

  // Broadcast message to all clients
  io.emit('new-message', message);

  console.log(`${client.role} ${client.name}: ${messageText}`);
});

// Handle WebRTC signaling for audio streaming
socket.on('webrtc-offer', (data) => {
  const client = connectedClients.get(socket.id);
  if (client && client.role === 'teacher') {
    socket.broadcast.emit('webrtc-offer', {
      offer: data.offer,
      senderId: socket.id
    });
  }
});
```

```
  socket.on('webrtc-answer', (data) => {
    if (data.targetId && connectedClients.has(data.targetId)) {
      io.to(data.targetId).emit('webrtc-answer', {
        answer: data.answer,
        senderId: socket.id
      });
    }
  });

  socket.on('webrtc-ice-candidate', (data) => {
    if (data.targetId && connectedClients.has(data.targetId)) {
      io.to(data.targetId).emit('webrtc-ice-candidate', {
        candidate: data.candidate,
        senderId: socket.id
      });
    } else {
      socket.broadcast.emit('webrtc-ice-candidate', {
        candidate: data.candidate,
        senderId: socket.id
      });
    }
  });

  // Handle disconnection
  socket.on('disconnect', () => {
    const client = connectedClients.get(socket.id);
    if (client) {
      console.log(`${client.role} ${client.name} disconnected`);

      if (client.role === 'teacher') {
        classroomState.isTeacherPresent = false;
        socket.broadcast.emit('teacher-left');
      }

      connectedClients.delete(socket.id);
      classroomState.participants = Array.from(connectedClients.values());

      io.emit('participants-updated', classroomState.participants);
    }
  });
});

// Cleanup function for old slides (call periodically)
```

```
function cleanupOldSlides() {
  const slidesDir = path.join(__dirname, 'slides');
  if (!fs.existsSync(slidesDir)) return;

  const dirs = fs.readdirSync(slidesDir);
  const cutoffTime = Date.now() - (24 * 60 * 60 * 1000); // 24 hours ago

  dirs.forEach(dir => {
    const dirPath = path.join(slidesDir, dir);
    try {
      const stats = fs.statSync(dirPath);
      if (stats.isDirectory() && stats.mtime.getTime() < cutoffTime) {
        fs.rmSync(dirPath, { recursive: true, force: true });
        console.log(`Cleaned up old slides directory: ${dir}`);
      }
    } catch (err) {
      console.error(`Error cleaning up ${dir}:`, err.message);
    }
  });
}

// Run cleanup every hour
setInterval(cleanupOldSlides, 60 * 60 * 1000);

// Error handling middleware
app.use((error, req, res, next) => {
  if (error instanceof multer.MulterError) {
    if (error.code === 'LIMIT_FILE_SIZE') {
      return res.status(400).json({ error: 'File too large. Maximum size is 50MB.' });
    }
  }
  res.status(500).json({ error: error.message });
});

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 Virtual Classroom Server running on http://localhost:${PORT}`);
  console.log(`📚 Open multiple tabs to test teacher/student interaction`);

  // Create necessary directories
  const dirs = ['uploads', 'slides', 'resources'];
  dirs.forEach(dir => {
    if (!fs.existsSync(dir)) {
      fs.mkdirSync(dir, { recursive: true });
    }
  });
});
```

Package.json

```
{
  "name": "low_bandwidth_demo",
  "version": "1.0.0",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "dependencies": {
    "canvas": "^3.2.0",
    "express": "^5.1.0",
    "multer": "^2.0.2",
    "pdf-img-convert": "^2.0.0",
    "pdf-poppler": "^0.2.1",
    "pdf-to-png-converter": "^3.7.1",
    "pdfjs-dist": "^5.4.149",
    "pptx2pdf": "^1.0.10",
    "sharp": "^0.34.3",
    "socket.io": "^4.8.1",
    "uuid": "^11.1.0",
    "ws": "^8.18.3"
  }
}
```