**API Documentation**

**Introduction**

This document provides comprehensive documentation for the Collaborative Document Editor's backend services. It covers the REST API for standard operations and the WebSocket events used for real-time features. The system uses Firebase for user authentication and data persistence.

**Base URL**

* **Development**: http://localhost:5000/api

**Authentication Mechanism (Firebase Auth)**

The system uses Firebase Authentication to manage users. On the frontend, the client signs in using a standard Firebase method (e.g., email and password). Firebase then issues a JSON Web Token (JWT) ID token. This token must be included in the Authorization header for all protected API requests.

**Header Format :**

Authorization: Bearer <firebase\_jwt\_id\_token>

The backend verifies this token using the Firebase Admin SDK on every request to authenticate the user and authorize access to resources.

**Socket Events (for real-time)**

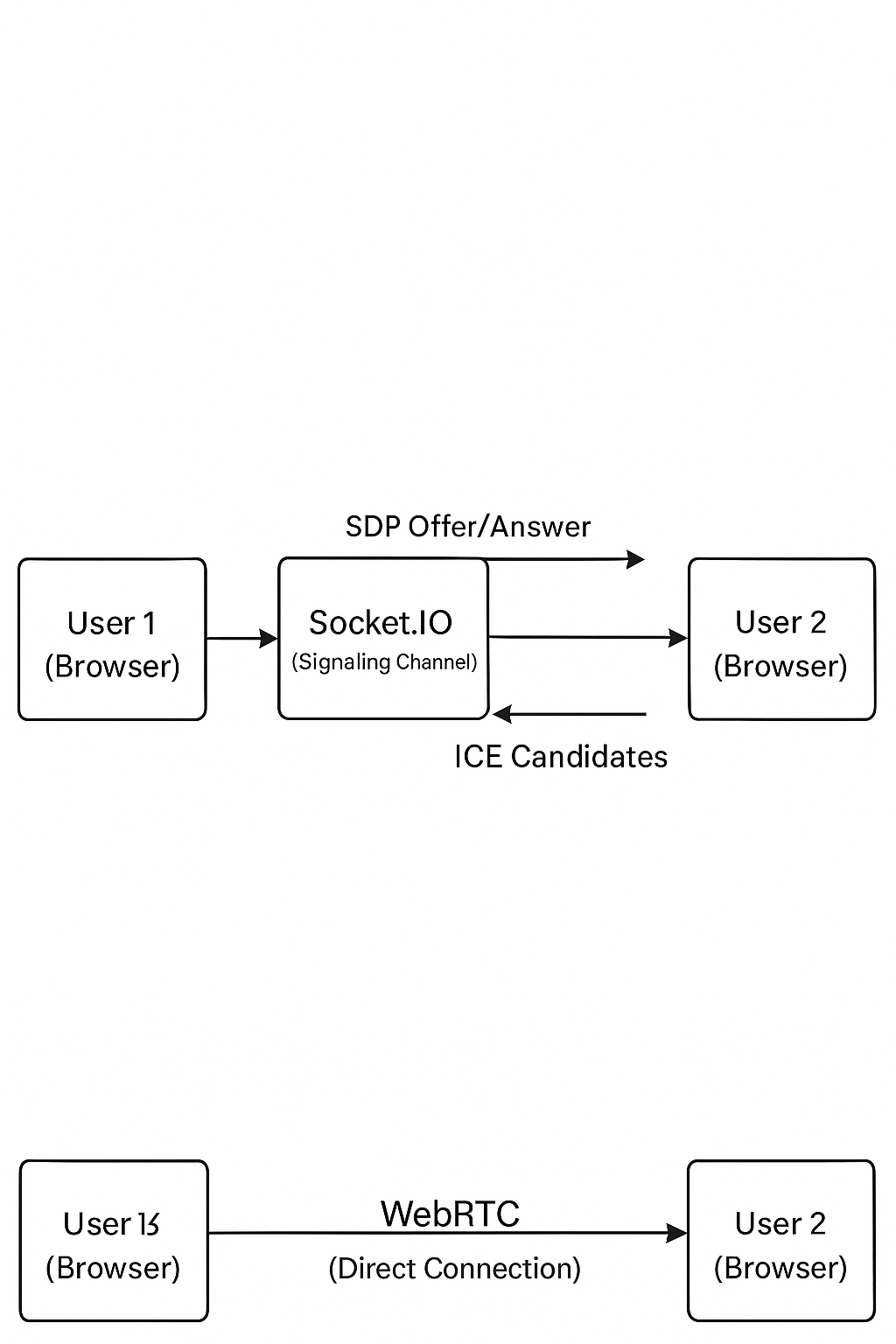
The Socket connection is managed by Socket.IO and is crucial for real-time collaboration. The connection must be authenticated by passing the user's Firebase JWT ID token upon connection.

|  |  |  |  |
| --- | --- | --- | --- |
| **Event Name** | **Direction** | **Description** | **Payload Example** |
| join-document | Client → Server | Client requests to join a specific document's room to receive real-time updates. | { "documentId": "doc\_123" } |
| leave-document | Client → Server | Client informs the server it is leaving a document's room. | { "documentId": "doc\_123" } |
| content-change | Client → Server | A user sends their edits to the server to be broadcast to other collaborators. | { "documentId": "doc\_123", "changes": {...} } |
| cursor-position | Client → Server | A user sends their cursor position to be shown to others in real-time. | { "documentId": "doc\_123", "position": 245 } |
| user-joined | Server → Client | Notifies clients in a room that a new user has connected. | { "userId": "user\_abc", "userName": "Jane Doe" } |
| user-left | Server → Client | Notifies clients that a user has disconnected. | { "userId": "user\_abc" } |
| content-updated | Server → Client | Broadcasts document changes from one user to all others in the room. | { "changes": {...}, "sourceUser": "user\_xyz" } |
| cursor-updated | Server → Client | Broadcasts a user's cursor position to all others. | { "userId": "user\_abc", "position": 245 } |

**WebRTC Signaling Flow**

WebRTC is used for direct peer-to-peer communication (e.g., for voice/video chat). The WebSocket server acts as the signaling channel to help peers find and connect with each other. This flow is managed via WebSocket events.

1. **Offer**: When User A wants to connect to User B, User A's client sends a webrtc-offer event to the server, specifying User B as the target.
2. **Forward Offer**: The server forwards this offer to User B via a webrtc-offer-received event.
3. **Answer**: User B's client receives the offer and generates an answer, which it sends back to the server with a webrtc-answer event, targeting User A.
4. **Forward Answer**: The server forwards this answer to User A via a webrtc-answer-received event.
5. **ICE Candidates**: As both clients discover network candidates (ways to connect), they send webrtc-ice-candidate events to the server, which forwards them to the other peer.
6. **Connection**: Once the offer/answer exchange is complete and ICE candidates are shared, a direct WebRTC peer-to-peer connection is established.



A diagram with colorful text

AI-generated content may be incorrect.

**Justification for my approach**

**🔹 1. What is SDP Offer ?**

**SDP (Session Description Protocol)** is used in **WebRTC** to describe media capabilities (like codec, IP, ports, etc.) between two peers. It works like this:

**✅ SDP Offer/Answer Model:**

* **Peer A** creates an **SDP Offer**:  
  It contains information about what kinds of media Peer A wants to send/receive.
* **Peer B** receives the offer and responds with an **SDP Answer**, saying what it agrees to.

This negotiation helps both peers agree on:

* Media formats (video/audio codec)
* IP address & port to connect
* Whether to send/receive audio/video/data

👉 **Example**:

{

"type": "offer",

"sdp": "v=0\no=- 4611736600794295656 2 IN IP4 127.0.0.1..."

}

**🔹 2. What are ICE Candidates?**

**ICE (Interactive Connectivity Establishment)** candidates are:

* Different **network paths** (IP:port combinations) a peer can use to connect.
* These include:
  + Host (local IP)
  + STUN (public IP via NAT traversal)
  + TURN (relay server fallback)

Peers exchange ICE candidates after the SDP negotiation to find the **best network path** to establish a **direct P2P** connection.

**🔹 3. Why use Socket.IO for WebRTC?**

WebRTC does **not** provide its own signaling mechanism. That means it **needs a signaling channel** to:

* Exchange SDP offers and answers
* Exchange ICE candidates

You use **Socket.IO** as that **signaling layer**. It sends the necessary WebRTC negotiation messages between clients via the server **before** the direct P2P connection is established.

**🔸 So, why not just use WebRTC alone?**

Because:

* WebRTC handles **media/data transmission**, but **not signaling**.
* You still need something (like **Socket.IO**) to handle signaling.

| **Purpose** | **WebRTC** | **Socket.IO** |
| --- | --- | --- |
| Send audio/video/data | ✅ | ❌ |
| Exchange connection info (SDP/ICE) | ❌ | ✅ |
| Peer-to-peer communication | ✅ | ❌ |
| Initial negotiation setup | ❌ | ✅ |

**✅ Summary**

* **WebRTC** = actual real-time connection.
* **Socket.IO** = helper tool for setting up the connection (signaling).
* **SDP Offer/Answer** = describe connection capabilities.
* **ICE Candidates** = potential network paths for peer connection.

**API Routes**

The following are the custom backend REST API endpoints.

**Authentication**

**POST /auth/register**

* **Description**: Registers a new user in the Firebase system.
* **Request Method**: POST
* **Request Body**:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| email | string | The user's email address. Must be unique. |
| name | string | The user's full name. |
| password | string | The user's password (min. 6 characters). |

* **Response (201 Created)**: Returns a confirmation message, the new user object, and a session token.
* **Status Codes**:
  + 201: User created successfully.
  + 400: Bad request (e.g., invalid email, weak password).
  + 409: Conflict (email already exists).

**Documents**

**GET /documents**

* **Description**: Retrieves all documents accessible to the authenticated user.
* **Request Method**: GET
* **Response (200 OK)**: Returns an object containing an array of Document objects. See the [Data Models](https://www.google.com/search?q=%23data-models) section for the Document object structure.
* **Status Codes**:
  + 200: Success.
  + 401: Unauthorized.

**POST /documents**

* **Description**: Creates a new document.
* **Request Method**: POST
* **Request Body**:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| title | string | The title of the new document. |
| content | string | The initial HTML content of the document. |
| isPublic | boolean | Whether the document is publicly accessible. |

* **Response (201 Created)**: Returns a confirmation message and the newly created Document object.
* **Status Codes**:
  + 201: Document created successfully.
  + 400: Bad request (e.g., missing title).
  + 401: Unauthorized.

**GET /documents/:id**

* **Description**: Retrieves a single document by its ID.
* **Request Method**: GET
* **URL Parameters**:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| id | string | The unique identifier of the document. |

* **Response (200 OK)**: Returns an object containing the full Document object.
* **Status Codes**:
  + 200: Success.
  + 401: Unauthorized.
  + 403: Forbidden (user does not have access).
  + 404: Not Found.

**PUT /documents/:id**

* **Description**: Updates a document's title or content.
* **Request Method**: PUT
* **URL Parameters**:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| id | string | The unique identifier of the document to update. |

* **Request Body**:

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| title | string | The new title for the document. |
| content | string | The new HTML content for the document. |

* **Response (200 OK)**: Returns a confirmation message and the updated Document object.
* **Status Codes**:
  + 200: Success.
  + 401: Unauthorized.
  + 403: Forbidden.
  + 404: Not Found.

**DELETE /documents/:id**

* **Description**: Deletes a document. Only the document owner can perform this action.
* **Request Method**: DELETE
* **URL Parameters**:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Type** | **Description** |
| id | string | The unique identifier of the document to delete. |

* **Response (200 OK)**:
* {
* "message": "Document deleted successfully"
* }
* **Status Codes**:
  + 200: Success.
  + 401: Unauthorized.
  + 403: Forbidden.
  + 404: Not Found.

**Data Models**

**User Object**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| userId | string | The unique Firebase UID. |
| email | string | The user's email address. |
| name | string | The user's full name. |

**Document Object**

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| id | string | The unique identifier for the document. |
| title | string | The title of the document. |
| content | string | The full content of the document in HTML format. |
| isPublic | boolean | Indicates if the document is accessible to anyone. |
| createdBy | string | The userId of the user who created the document. |
| collaborators | Array<string> | An array of userIds who have access to the document. |
| createdAt | string | ISO 8601 timestamp of when the document was created. |
| updatedAt | string | ISO 8601 timestamp of the last update. |