Chatterly

Architecture Document

Written By: Mahmoud Bakr **Date:** March 2026

Table of Contents

Background

This document describes the architecture of Chatterly, a real-time team messaging and collaboration platform built for developers and teams who need a fast, reliable, and unified communication tool.

Modern teams rely on multiple fragmented tools to communicate — some use email for decisions, others use different group chats, and real-time calls happen across disconnected platforms. This creates a problem: context is lost, notifications are scattered, and there is no single place where a team's conversation history, presence, and collaboration are unified.

Chatterly solves this by providing a single, real-time platform that combines persistent messaging (organized into channels and direct messages), online presence tracking, and peer-to-peer voice and video calls. Users can see who is online, exchange messages in shared channels or private groups, react to messages, and initiate calls — all from one interface.

From a business perspective, Chatterly is designed as a showcase application demonstrating production-grade system design, scalability decisions, and real-time architecture. It is built by a solo developer with a focus on learning and demonstrating system architecture principles at scale. The target scale goal is supporting up to one million concurrent WebSocket connections, following the same infrastructure patterns used by applications that have scaled to tens of millions of users.

The authentication system, user model, and all core data models are finalized and implemented. The platform supports full self-registration via a public `POST /auth/register` endpoint, JWT-based login, and logout with Redis-backed token revocation.

This document describes the system architecture of Chatterly.

The architecture comprises technology and modeling decisions that will ensure the final product, assuming the architecture is followed, will be fast, reliable, and easy to maintain. The document outlines the thought process for every aspect of the architecture and clearly explains why specific decisions were made.

It is extremely important for the development team to closely follow the architecture depicted in this document. In any case of doubt, please consult the Software Architect.

Requirements

Functional Requirements

- Register and authenticate users securely using JWT-based authentication
- Organize conversations into three types: channels (public), groups (private), and direct messages
- Send and receive real-time messages within conversations
- Support message threading via parent-message references
- Support soft deletion of messages (deleted content hidden, record preserved)
- Allow users to react to messages with emoji reactions
- Track and broadcast user online/offline presence in real time
- Initiate and manage peer-to-peer voice and video calls between users
- Support a full call lifecycle: calling, ringing, active, ended, declined, and missed states
- Expose a REST API for all data mutations and queries
- Broadcast real-time events (messages, presence, call signals) over WebSocket connections

Non-Functional Requirements

The following non-functional requirements were defined based on the product's scale goals and agreed upon as the architectural targets.

Requirement	Target
REST API reads	< 50ms (Redis cache hit)
REST API writes	< 100ms (PostgreSQL write + Redis pub/sub)
WebSocket delivery	< 30ms (Redis O(1) fan-out)
Auth check latency	< 5ms (Redis GET for JWT denylist)
Throughput	~50,000 requests/second (horizontal Puma scaling)
Concurrent WS	~1,000,000 connections (Redis pub/sub, Puma threads)
Message loss	0% (PostgreSQL ACID transactions)
SLA	Platinum — high availability, zero data loss
Data volume	Partitioned messages table (monthly), handles hundreds of millions of rows

---

Executive Summary

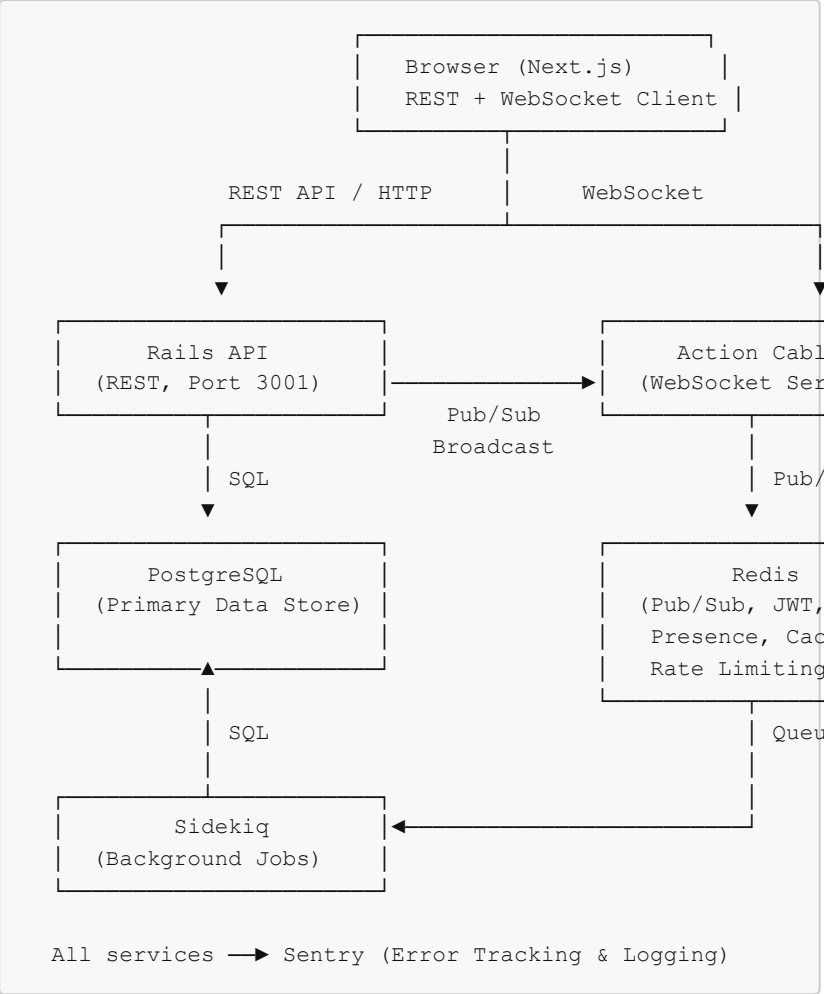
This document describes the architecture of Chatterly, a modern real-time messaging and collaboration platform. The platform allows users to communicate through persistent channels and direct messages, track team presence, and conduct peer-to-peer voice and video calls.

When designing the architecture, a strong emphasis was placed on two major features:

- The application should be **reliable** — no messages lost, no dropped connections, consistent state across all clients
- The application should be **extremely fast** — real-time delivery measured in milliseconds, not seconds

To achieve these goals, the architecture is based on a hybrid communication model: REST API for data mutations and queries, and WebSocket connections for real-time push events. The two concerns are kept strictly separate, each optimized for what it does best.

Here is a high-level overview of the architecture:



As shown in the diagram, the application is comprised of four separate, focused components — Rails API, Action Cable, Sidekiq, and Next.js — each with a well-defined role. They communicate using standard protocols, and all persistent state lives in PostgreSQL and Redis, making every component fully stateless and safely replaceable.

This architecture, built on Ruby on Rails 8 and Next.js, follows modern best practices in distributed system design, ensuring the platform can scale to millions of users while remaining straightforward to develop and maintain.

Overall Architecture

Services

The architecture is comprised of the following services:

Rails API — The primary HTTP service. It handles all REST API requests: user authentication, conversation management, message creation, reactions, and call lifecycle management. It is the only service that performs data mutations, writing to PostgreSQL as the source of truth. After a write, it triggers real-time broadcasts via Redis pub/sub so that Action Cable can fan out the event to all connected clients. The Rails API does not handle WebSocket connections directly — that is the responsibility of Action Cable.

Action Cable — The real-time WebSocket server. It maintains persistent connections with all connected clients and pushes events to them as they occur. It does not perform data mutations; its role is purely push delivery. It subscribes to Redis pub/sub streams and forwards events to the appropriate WebSocket clients. Three channels are defined: ConversationChannel (message events), PresenceChannel (online/offline events), and CallChannel (WebRTC signaling).

Sidekiq — The background job processor. It polls a Redis-backed job queue and executes asynchronous tasks that do not need to happen in the request-response cycle. The primary example is the missed call job: when a call receives no answer within 30 seconds, Sidekiq updates the call session status to `missed`. This pattern keeps the Rails API fast and non-blocking for time-sensitive background work.

Next.js Client — The frontend application served to the browser. It communicates with the Rails API over HTTP for data operations and maintains a WebSocket connection to Action Cable for real-time event delivery. It also manages the peer-to-peer WebRTC connection for voice and video calls, using Action Cable as the signaling channel.

Scaling

This architecture allows each service to be scaled independently, because each service has a single, focused responsibility. The Rails API can be scaled horizontally by adding more Puma worker processes or additional server instances behind a load balancer. Action Cable scales through Redis pub/sub: regardless of which server a client is connected to, all servers subscribe to the same Redis channels and receive the same broadcast events. Sidekiq scales by adding more worker processes, each pulling from the same Redis job queue. PostgreSQL scales through a read replica (pre-wired in `database.yml` via `DATABASE_REPLICA_URL`) that can be enabled with zero code changes.

All services are stateless by design. No in-memory session state is held between requests. JWT tokens are verified against a Redis denylist, and user presence is tracked in Redis with TTL-based auto-expiry. This means any service instance can be terminated or replaced without data loss.

Messaging

The services communicate with each other using different messaging methods, each chosen based on the specific requirements of that interaction:

Rails API exposes REST API over HTTP. This is the standard for client-server communication, supports clear request/response semantics, and is compatible with all HTTP clients. It is the natural choice for data mutations and queries where the client needs an immediate, confirmed response.

Action Cable uses the WebSocket protocol. REST API is a poor fit for server-to-client push — it requires polling. WebSocket maintains a persistent, bidirectional connection, allowing the server to push events to clients at the moment they occur, without the client having to ask. This is essential for real-time messaging and presence.

Sidekiq does not expose a classic API. It polls a Redis-backed job queue. This is appropriate because background jobs do not require synchronous handling — the Rails API enqueues a job and moves on immediately. The Queue also adds reliability: if Sidekiq restarts, the jobs remain in the queue and are processed when it comes back up.

Redis pub/sub is used for Rails API → Action Cable communication. When the Rails API writes a new message to the database, it publishes an event to a Redis channel. All Action Cable server processes subscribe to that channel and receive the event simultaneously, then push it to their connected WebSocket clients. This is how real-time fan-out works at scale without coupling the Rails API to specific WebSocket connections.

Services Drill Down

Rails API

Role

The Rails API is the core HTTP service of Chatterly. It is the single point of entry for all REST API requests from the Next.js client. Its responsibilities include authenticating users, managing conversations and their members, creating and soft-deleting messages, handling emoji reactions, and managing the call session lifecycle.

The Rails API is strictly responsible for data mutations and synchronous query responses. It does not maintain WebSocket connections and does not push events directly to clients. Instead, after every write operation, it publishes a broadcast event to Redis, which Action Cable picks up and delivers to connected clients.

This separation of concerns keeps the Rails API lean, predictable, and easy to scale.

Technology Stack

The Rails API is built on **Ruby on Rails 8.1 in API mode**, running on **Ruby 3.4.4**. Rails was selected for its convention-over-configuration approach, its mature ActiveRecord ORM, and its built-in ecosystem for authentication, authorization, and serialization. Running in API mode strips the middleware stack down to only what is needed for JSON API responses.

PostgreSQL 17 is the primary data store. It was chosen for its ACID compliance, native support for range-based table partitioning (used for the messages table), and its reliability as a CP (Consistent + Partition-tolerant) system in CAP theorem terms. No messages are ever lost due to partial writes.

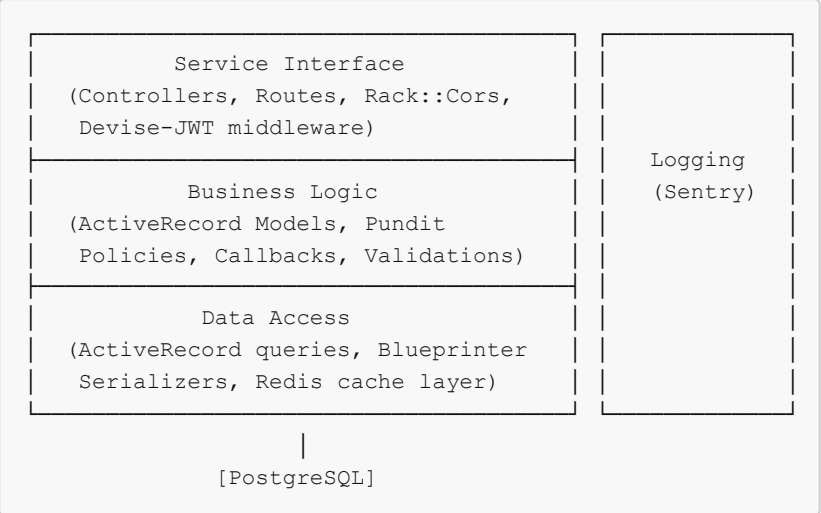
Devise 4.9 + devise-jwt 0.13 handle authentication. JWT tokens are issued on login and passed in request headers, making the authentication stateless and compatible with cross-origin requests from the Next.js frontend.

Pundit 2.5 handles authorization through Policy Objects. Each model has a corresponding policy class (`ConversationPolicy`, `MessagePolicy`, `CallSessionPolicy`) that encapsulates access rules, keeping authorization logic out of controllers.

Blueprinter is used for serialization. It converts ActiveRecord objects into clean JSON structures, controlling exactly what fields are exposed. The same serializer is reused in both HTTP responses and WebSocket broadcasts, ensuring consistency.

PgBouncer (via Supabase port 6543) is used as a connection pooler in production, keeping the number of live PostgreSQL connections bounded under horizontal Puma scaling.

Architecture



Every HTTP request flows top to bottom through this stack:

The **Service Interface** layer receives the request, checks CORS headers, decodes the JWT token via Devise middleware, and routes the request to the correct controller action. Controllers contain as little logic as possible — they extract parameters and delegate immediately to the Business Logic layer.

The **Business Logic** layer is where the application's rules live. ActiveRecord models enforce validations, run callbacks (including `after_create_commit` for broadcasting), and apply scopes. Pundit policies enforce authorization rules before any data is read or written.

The **Data Access** layer translates business operations into database queries via ActiveRecord. Blueprinter serializers are invoked here to convert query results into clean JSON before they are returned to the controller for rendering.

Logging (Sentry) is a cross-cutting concern accessible by all layers. Every exception is captured and reported automatically by the Sentry Rails integration.

Implementation Instructions

- Controllers must contain no business logic. Their only jobs are: authenticate, authorize, call the model or service, and render the serialized result.
- Use `authorize @resource` (Pundit) in every controller action that touches a resource. Do not skip authorization.
- Use Blueprinter serializers consistently. Never call `.to_json` directly on an ActiveRecord object — always go through the serializer.
- Use `after_create_commit` and `after_update_commit` callbacks on models to trigger Action Cable broadcasts via Redis. Do not broadcast from controllers.
- Use `prepared_statements: false` and `advisory_locks: false` in `database.yml` when connecting through PgBouncer (transaction pooling mode does not support these features).

Action Cable

Role

Action Cable is the real-time WebSocket service of Chatterly. Its sole responsibility is to maintain persistent connections with clients and push events to them as they occur. It does not write data to the database and does not make business logic decisions — it is a delivery layer.

Clients subscribe to one or more channels upon connecting. When a relevant event is published to Redis by the Rails API, Action Cable receives it via pub/sub and forwards it to all

subscribed clients on the appropriate channel. This architecture allows real-time fan-out across thousands of connected clients without any polling.

Technology Stack

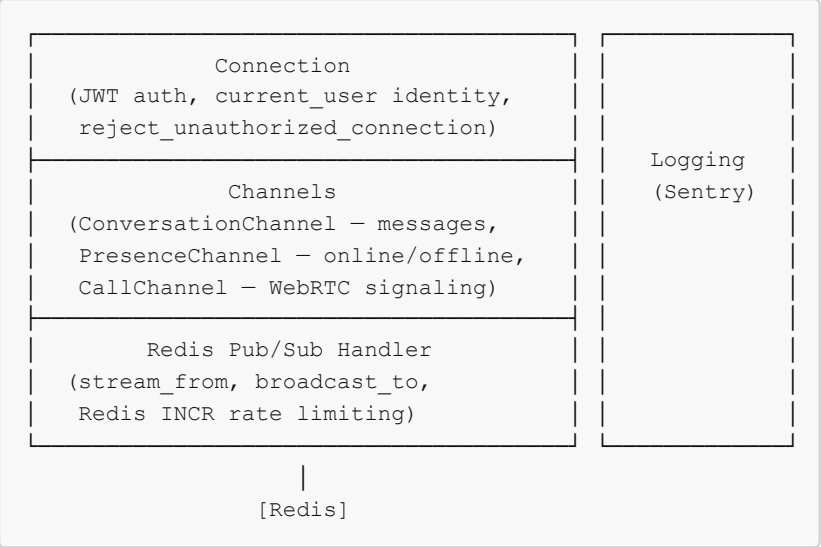
Action Cable is built into **Ruby on Rails** and runs within the same Rails process as the API in development. In production, it can be deployed as a separate Puma process to isolate WebSocket traffic from HTTP traffic.

The **Redis adapter** is used for Action Cable's pub/sub backend. Redis was chosen over Solid Cable (PostgreSQL-based) because of its O(1) pub/sub fan-out performance, which is essential at the target scale of one million concurrent WebSocket connections. Solid Cable uses PostgreSQL polling and becomes a bottleneck at high WebSocket concurrency.

Redis logical database /0 is dedicated to Action Cable pub/sub streams, keeping it isolated from Sidekiq queues, JWT denylist, presence data, and caching, which each have their own logical databases.

JWT authentication is verified on WebSocket connection using the same `devise-jwt` gem as the REST API, ensuring that only authenticated users can establish WebSocket connections.

Architecture



The **Connection** layer authenticates every WebSocket handshake. It decodes the JWT token from the query string, checks it against the Redis denylist, and sets `current_user`. If authentication fails, the connection is rejected before any channel subscription is allowed.

The **Channels** layer contains the three channel classes that clients subscribe to. `ConversationChannel` streams message events for a specific conversation. `PresenceChannel` broadcasts online/offline state changes for all users globally. `CallChannel` relays WebRTC signaling messages (offers, answers, ICE candidates) between specific users identified by their user ID streams.

The **Redis Pub/Sub Handler** layer manages the underlying Redis stream subscriptions (`stream_from`) and rate limiting logic. Rate limiting is implemented using Redis `INCR` with a TTL expiry, preventing clients from flooding channels with events.

Logging (Sentry) is a cross-cutting concern. WebSocket errors and connection lifecycle events are captured automatically.

Implementation Instructions

- Every channel action that receives data from the client must call `rate_limit!` before processing. This is defined on `ApplicationCable::Channel` and uses Redis INCR+TTL.
- Channel membership and authorization must be checked in `subscribed` before calling `stream_from`. A non-member must never receive another conversation's events.
- Do not query the database inside channel relay actions (e.g., `send_signal` in `CallChannel`). These are hot paths — relay data directly from the client to the target stream without a DB round-trip.
- Once Pundit policies are implemented (Phase 5), refactor channel membership guards to delegate to the appropriate policy: `ConversationChannel#member?` → `ConversationPolicy` , `CallChannel#participant?` → `CallSessionPolicy` .

Sidekiq

Role

Sidekiq is the background job processing service of Chatterly. It handles work that does not need to happen synchronously in the request-response cycle. Rather than exposing an API, it polls a Redis-backed job queue for work to do.

The primary use case in Chatterly is the **missed call job**: when a call session is initiated but the callee does not respond within 30 seconds, a Sidekiq job fires and transitions the call session status from `ringing` to `missed`. This logic cannot live in the Rails API request cycle because it is time-delayed — it must be scheduled, not triggered by an incoming request.

This design keeps the Rails API fast and non-blocking, while ensuring that time-sensitive state transitions happen reliably in the background.

Technology Stack

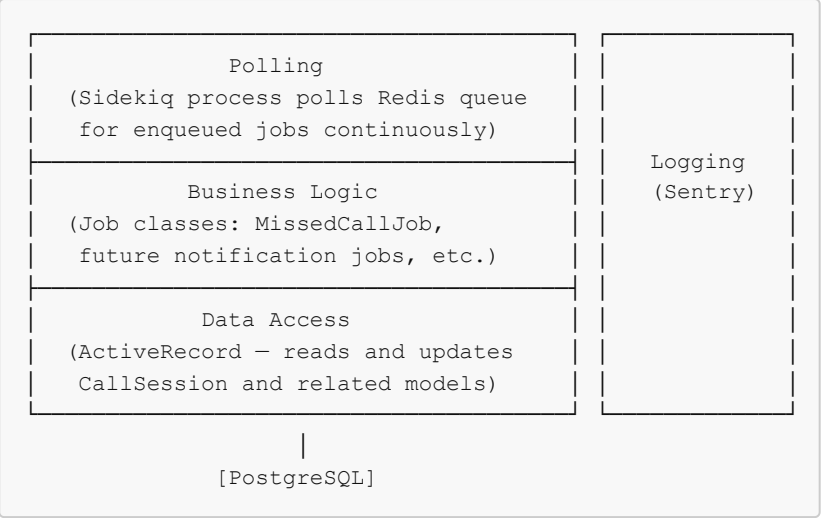
Sidekiq 7 is used as the background job framework. It was selected over Solid Queue (Rails 8's built-in queue backend) because Sidekiq uses Redis as its queue store, which matches Chatterly's existing Redis infrastructure and provides better performance at the target scale.

Redis logical database /1 is dedicated to Sidekiq job queues, isolated from the Action Cable pub/sub namespace.

Sidekiq workers run in a separate process from the Rails API. In development, `bundle exec sidekiq` starts the worker process. In production, it runs as a separate service on the hosting platform.

Jobs are plain Ruby classes that include `Sidekiq::Worker`. They have access to the full Rails environment, including ActiveRecord models, and can read from and write to PostgreSQL.

Architecture



The **Polling** layer is managed by Sidekiq itself. The Sidekiq process connects to Redis and continuously polls the job queue. When a job is available, it is dequeued and passed to the appropriate worker class. Sidekiq handles concurrency, retries on failure, and dead-letter queuing for jobs that exhaust retries.

The **Business Logic** layer contains the job classes. Each job class has a single `perform` method with clearly defined inputs (typically IDs, never full ActiveRecord objects). The `MissedCallJob` receives a `call_session_id`, looks up the session, and transitions it to `missed` if it is still in the `ringing` state at the time of execution.

The **Data Access** layer uses ActiveRecord to read from and write to PostgreSQL. No raw SQL is used — all queries go through the model layer, benefiting from validations and callbacks.

Logging (Sentry) is a cross-cutting concern. Sidekiq integrates with Sentry automatically to capture job failures and exceptions.

Implementation Instructions

- Job arguments must be simple scalar types (strings, integers). Never pass ActiveRecord objects as job arguments — serialize to an ID and reload inside `perform`.
- Always guard state before mutating. The `MissedCallJob` must check that the call session is still `ringing` before

transitioning to `missed`. The job may run after the call has already been answered or declined.

- Schedule jobs using `perform_in(30.seconds, call_session_id)` at the moment the call session is created with `ringing` status.
- Every step in a job must be logged. Since there is no request context or UI for background jobs, logging is the only way to trace what happened during a job execution.

Next.js Client

Role

The Next.js Client is the frontend application that runs in the user's browser. It is the only service that the end user directly interacts with. It is responsible for rendering the full user interface, managing client-side state, communicating with the Rails API over HTTP, maintaining a real-time WebSocket connection to Action Cable, and establishing peer-to-peer WebRTC connections for voice and video calls.

The Next.js Client does not store any business data persistently — it reads from the Rails API and reflects the current state of the server. Its local state (Zustand stores) is ephemeral and is rebuilt from API data on each session.

Technology Stack

Next.js (App Router) is the frontend framework. It was selected for its file-based routing, server component support for fast initial page loads, and its first-class TypeScript support.

Zustand manages client-side state. It was selected over Redux for its minimal API surface, absence of boilerplate, and the ability to access and update stores from any component without Provider wrappers.

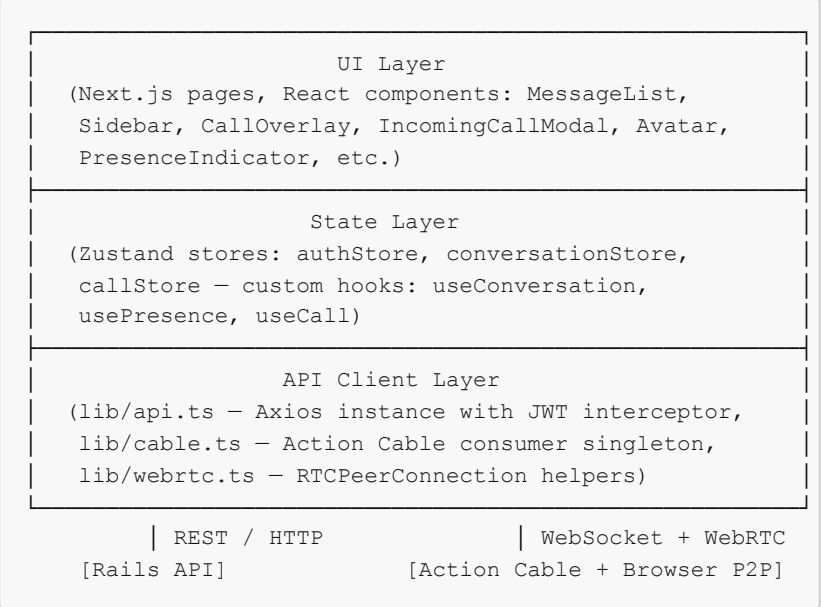
Tailwind CSS handles all styling through utility classes applied directly in JSX, eliminating context-switching between component and stylesheet files.

Axios is the HTTP client for communicating with the Rails API. An Axios instance is configured with a base URL and an interceptor that attaches the JWT token from local storage to every request header.

Action Cable JS (the official Rails WebSocket client) maintains the persistent WebSocket connection to the Action Cable server. A singleton consumer is created once in `lib/cable.ts` and shared across all channel subscriptions.

WebRTC (browser-native API) is used for peer-to-peer voice and video. The Rails Action Cable server handles signaling (SDP offer/answer and ICE candidate exchange). The actual audio and video streams flow directly between browsers without passing through the server.

Architecture



The **UI Layer** contains all React components and Next.js pages. Components are pure rendering — they read from Zustand stores and call store actions. No direct API calls are made from components; all data fetching is delegated to the State Layer via hooks.

The **State Layer** contains Zustand stores and custom hooks. `authStore` manages the authenticated user and JWT token. `conversationStore` holds the list of conversations, the active conversation, and its message history. `callStore` tracks the

current call state, participants, and mute/camera status. Custom hooks (`useConversation`, `usePresence`, `useCall`) bridge the Zustand stores with the API Client Layer, subscribing to Action Cable channels and updating store state when WebSocket events arrive.

The **API Client Layer** contains the low-level communication utilities. `lib/api.ts` provides the configured Axios instance. `lib/cable.ts` creates and exports the single Action Cable consumer used by all channel subscriptions. `lib/webrtc.ts` provides helpers for setting up `RTCPeerConnection`, generating SDP offers/answers, and handling ICE candidate exchange during call signaling.

Implementation Instructions

- The API Client Layer must never be called directly from UI components. All data operations must go through the State Layer (hooks and stores) to keep components pure and testable.
- The Action Cable consumer in `lib/cable.ts` must be a singleton. Do not create a new consumer per component — one consumer manages all channel subscriptions for the session.
- WebRTC peer connections must be created in `useCall` hook only, not inside components. The hook manages the full connection lifecycle: creation, signaling, ICE, and teardown on call end.
- JWT tokens must be stored in memory (Zustand store) or `httpOnly` cookies — not in `localStorage` — to prevent XSS token theft. The Axios interceptor reads from the store, not from `localStorage`.
- All WebSocket event handlers (`received` callbacks) must be idempotent. Events may occasionally be delivered more than once; the state update logic must handle duplicates safely.