ChatGPT

🏗️ Plan de Arquitectura Frontend para Sistema de Chat (Tipo Slack)

Este documento detalla la planificación **previa** y las decisiones tecnológicas para construir un frontend **profesional, robusto y mantenible** para un sistema de mensajería/chat (similar a Slack, o mejor). Está alineado con el backend existente **UTalk** (mensajería multicanal con CRM, campañas, chatbot, etc.), por lo que aprovecha al máximo sus capacidades [1] [2]. El objetivo es usar las **mejores tecnologías disponibles (futuras y disruptivas, no solo las más populares)**, pensando a 10 años adelante en rendimiento y mantenibilidad.

A continuación se presenta una **checklist secuencial** de todo lo necesario **antes de escribir el primer componente**, con las decisiones tomadas en cada punto y su justificación. Estas indicaciones servirán al equipo como guía paso a paso para preparar el proyecto frontend **al nivel de un backend enterprise**.

1. Definición de requerimientos y alcance

- [ ] **Módulos principales del MVP**: El alcance inicial abarca:
- **Autenticación**: Pantalla de **login** (y registro si aplica). Uso de email/contraseña con JWT, alineado con la API de autenticación del backend [3] [4]. Incluye gestión de sesión (tokens de acceso y refresco).
- **Chat/Conversaciones**: Vista principal tipo Slack con listado de conversaciones (canales o contactos) y área de mensajes. Soporta listar conversaciones asignadas o del usuario [5], iniciar nuevas conversaciones, y cargar el historial de mensajes con paginación [6] [7].
- **Mensajes**: Envío y recepción en tiempo real de mensajes de texto (y multimedia en futuras iteraciones). Incluye mostrar estado del mensaje (enviado, entregado, leído) y funcionalidades tipo *typing indicator* ("Usuario está escribiendo..."), entrega y lectura en vivo.
- **Contactos/Clientes**: (Para versión completa) módulo de contactos/CRM integrado, mostrando detalles del cliente/contacto asociado a cada conversación [8]. En MVP podría simplificarse a mostrar nombre del contacto en la conversación.
- **Perfil de usuario**: Página para ver/editar perfil (nombre, contraseña, etc.), utilizando los endpoints `/api/auth/profile` [9] y cambio de contraseña [10].
- [ ] **Features clave del MVP**:
- **Mensajería en tiempo real** (texto inicialmente): Al enviar un mensaje, debe aparecer instantáneamente en la interfaz de todos los clientes conectados a esa conversación [11] [12]. Nuevos mensajes entrantes se muestran sin refrescar la página.
- **Notificaciones**: Si el usuario está inactivo o la ventana en segundo plano, recibir notificaciones (ej. push del navegador) de mensajes nuevos. En la interfaz, usar notificaciones tipo *toast* para avisos breves (ej. "Mensaje enviado" o errores).
- **Indicadores de estado**: Mostrar qué usuarios están en línea o su estado (el backend tiene eventos de `USER_ONLINE` / `OFFLINE` [13]). Mostrar indicadores de escritura en conversaciones (evento `typing`).
- **Seguridad y roles**: Soportar distintos roles de usuario (por ahora **admin** y **agente**). Un admin podría ver todas las conversaciones y métricas, un agente solo las asignadas o propias [14]. (Un **cliente** externo no tendrá interfaz web propia en MVP, ya que interactúa vía canales externos

como WhatsApp/SMS, pero podríamos en el futuro ofrecer un portal de cliente o widget web de chat).
- **Multicanal**: Preparar la UI para diferenciar mensajes según canal (WhatsApp, SMS, email) con íconos o etiquetas, aunque en MVP quizá solo se enfoque en chat interno. El backend soporta múltiples canales [15], por lo que la arquitectura frontend debe ser flexible para añadir vistas de campaña, chatbot y otras pestañas en versiones posteriores.
- [ ] **Alcance técnico**:
- La aplicación será **web (SPA)** con capacidad PWA (instalable, con offline básico y push). No se desarrollará app móvil nativa en esta fase, pero el PWA dará experiencia similar a app.
- Debe escalar a miles de usuarios concurrentes en tiempo real, manteniendo rendimiento óptimo. El backend ya es enterprise (usa Node.js + Socket.IO optimizado para miles de conexiones [16] [17]), por lo que el frontend debe manejar eficientemente actualizaciones frecuentes (virtual DOM o reactividad eficiente, etc.).
- Compatibilidad: Soportar últimos navegadores modernos. Uso de estándares ES2025+ (transpilados según necesite). Considerar accesibilidad (WCAG) desde el inicio – teclas de navegación en chat, marcas aria, etc.

2. Selección y documentación del *stack* tecnológico

Decidimos un *stack* de frontend **moderno y de alto rendimiento**, con énfasis en TypeScript, reactividad eficiente y herramientas de última generación:

- [ ] **Framework principal**: **SvelteKit** (con Svelte 4). SvelteKit ofrece una arquitectura moderna sin *virtual DOM* y compila componentes altamente optimizados, resultando en cargas más rápidas y menor huella de JavaScript comparado con React/Next [18] [19]. Su enfoque reactivo y sintaxis concisa permiten desarrollar más rápido y con menos código (**DX** sobresaliente). En 2025, Svelte y su ecosistema se consideran muy **future-proof** por su rendimiento y simplicidad – de hecho, sitios SvelteKit suelen superar a equivalentes en Next.js en métricas como First Contentful Paint [18]. *Alternativas evaluadas*: **Next.js 13** (React) por su solidez y comunidad. Next es "seguro" y cuenta con React Server Components, pero priorizamos SvelteKit por ser más ligero y "elegante" a largo plazo [19]. *Nota:* El equipo deberá capacitarse en SvelteKit si es nuevo, pero la curva es manejable dado que la sintaxis es intuitiva y similar a JavaScript puro [20].
- [ ] **Lenguaje**: **TypeScript** (estricto). TS es imprescindible para un proyecto grande en 2025 – prácticamente "ya no es opcional" [21]. Brinda tipado estático robusto, evitando muchos bugs en desarrollo y facilitando el mantenimiento a gran escala [22] [23]. Con TS tendremos autocompletado, refactors seguros y documentación viva en el código, lo que acelera el desarrollo en equipo. Además, SvelteKit soporta TS de forma nativa en sus plantillas.
- [ ] **Sistema de estilos (CSS)**: **Tailwind CSS** (utility-first). Esta elección permite diseñar rápidamente con clases utilitarias, garantizando consistencia de estilo y un CSS final optimizado (sin clases no usadas gracias a *purge*). Según encuestas, Tailwind destaca como el framework de CSS que los desarrolladores *"están felices de seguir usando"* [24]. Al ser utility-first, facilita aplicar el **diseño custom** requerido (Slack tiene estilo propio, no genérico) directamente en los componentes, sin necesidad de sobreescribir estilos predefinidos. Empresas líderes como OpenAI, GitHub, Shopify ya confían en Tailwind [25]. *Ventaja:* genera UIs consistentes con menos CSS manual, y su enfoque modular por componente encaja perfecto con frameworks modernos [26]. Como contrapartida, puede producir HTML con muchas clases, pero el mantenimiento global resulta más sencillo que con CSS tradicional [27].
- [ ] **Biblioteca de UI**: **Shadcn/UI** (componentes construidos sobre Tailwind + Radix UI). Esta librería (relativamente nueva, 2023) proporciona componentes accesibles y altamente personalizables sin el peso de un estilo predefinido pesado. **Shadcn** es *mucho más ligera que Material UI* y sus componentes se adaptan fácilmente a cualquier diseño [28], lo que viene

perfecto para lograr una interfaz tipo Slack (personalizada) sin reinventar todo. Básicamente, nos permite *"bootstrapear"* un diseño propio usando piezas listas (inputs, modales, menús, etc.) ya estilizadas con Tailwind. Al apoyarse en **Radix UI** bajo el capó, garantiza accesibilidad (ARP) y comportamientos UI consistentes. *Alternativas:* **Chakra UI** (popular, accesible) o **Material UI** (robusta pero pesada y con opinion de diseño Material). Optamos por Shadcn porque equilibra simplicidad y flexibilidad, con diseño moderno y mínimo bloat [29]. Cabe notar que al ser nueva, su comunidad es más pequeña, pero dado que se compone de Tailwind+Radix (tecnologías sólidas), no supone riesgo alto.

- [ ] **Gestión de estado global**: Adoptaremos una combinación moderna:
- **Svelte stores** para estado global sencillo (SvelteKit lo incluye de serie). En Svelte, las *stores reactivas* cumplen el rol de Context/Redux sin boilerplate: son simples objetos reactivos a los que los componentes se suscriben. Esto cubre cosas como el usuario logueado, estado de tema, etc., de forma muy fácil. Su mecanismo es eficiente (actualiza solo los suscritos) y escalable para bastantes casos, reduciendo la necesidad de librerías externas.
- **TanStack Query (React Query)** para estado de servidor (datos async): Utilizaremos su variante para Svelte (svelte-query) o directamente llamadas con *load functions*, pero con un enfoque similar a React Query. Esta librería es el estándar en 2025 para manejar datos asíncronos y cache de API [30]. Ventajas: *fetching* centralizado, cacheo automático, refetch en segundo plano, control de estados de carga/error, y hasta paginación out-of-the-box. Esto viene perfecto para cargar conversaciones, mensajes, etc., manteniéndolos actualizados sin múltiples llamadas redundantes [31] [32]. Con TanStack Query reduciremos la necesidad de Redux para datos remotos [33].
- **Zustand** (o **Context API** simple) para estado global UI puntual: Aunque Svelte stores cubrirán esto, mencionamos Zustand ya que en React es preferido sobre Redux en 2025 por su simplicidad y performance [34]. En nuestro caso, Svelte stores son análogas a un Zustand incorporado. No usaremos Redux Toolkit a menos que el dominio crezca en complejidad extrema; Redux sigue siendo poderoso pero su sobrecarga no se justifica en este frontend (priorizamos soluciones más simples) [35].
- [ ] **Librería de formularios**: **React Hook Form** (en SvelteKit podríamos usar *svelte-forms-lib* o directamente validar manual, pero posiblemente integremos React Hook Form a través de un wrapper o usemos su filosofía). **¿Por qué RHF?** Es extremadamente performante y ligero, evitando rerenders innecesarios al manejar formularios con refs no controladas [36]. En React ha reemplazado prácticamente a Formik; de hecho RHF tiene ~2x descargas que Formik en 2024 y un bundle menor [37]. Nos interesa especialmente para el **formulario de login** (y futuros formularios como perfil), ya que asegurará validación instantánea y manejo de errores sencillo. Ofrece integración con esquemas (p.ej. YUP o Zod) para validar campos antes de enviar. Formik, en cambio, implica más re-renders y código boilerplate [38] [39]. Así que RHF nos da **simplicidad + rendimiento** (menor código y validación robusta).
- [ ] **Sistema de rutas**: Usaremos el **enrutamiento de SvelteKit** basado en filesystem. Es decir, cada página vista corresponde a un archivo en `src/routes`. Esto nos da **SSR opcional** (por ejemplo, podríamos SSR la página de login para optimizar First Load) y también permite *routing* SPA cliente. No necesitaremos algo tipo React Router; SvelteKit provee navegación optimizada (prefetching de enlaces, etc.). En caso de haber optado por Next.js, hubiéramos usado el nuevo **App Router** de Next 13 (basado también en estructura de carpetas, con ventajas de RSC). Ambas opciones brindan segmentación de rutas y *layouts* anidados, útiles para: layout general (sidebar de conversaciones + header) con sub-rutas para cada conversación, etc. Documentaremos las rutas principales (p. ej. `/login`, `/conversations/:id` para chat abierto, etc.).
- [ ] **Cliente HTTP**: **Axios** para llamadas REST. Aunque *fetch* nativo ha mejorado, Axios aporta comodidades importantes: manejo automático de JSON (convierte responses a objeto), mejores mensajes de error, y sobre todo **interceptores** [40]. Con interceptores podemos, por ejemplo, agregar el token JWT a **todas** las peticiones fácilmente, o refrescar tokens en caso de 401

automáticamente. Esto nos ahorra escribir ese boilerplate en cada llamada. Axios también es promisorio y manejable en TS (tiene tipos incluidos). Dado que el backend expone una API REST robusta [1], Axios encaja bien como wrapper. Notar que Axios en sí hoy en día usa fetch internamente (es casi un "wrapper" sobre fetch) pero trae ~13KB gzips extras [41]; asumimos ese pequeño costo por la conveniencia que añade. Si quisiéramos minimizar dependencias, podríamos usar *fetch* con un wrapper propio, pero aprovecharemos Axios para ir más rápido en desarrollo.

- [ ] **Comunicación en tiempo real**: **Socket.IO client** (v4). El backend implementa un avanzado **Enterprise Socket Manager** basado en Socket.IO [42] [43], por lo que la mejor opción es usar la misma librería en el frontend para compatibilidad total. Esto nos permite autenticación de sockets con JWT durante el handshake (que ya está previsto en backend) [44] [45]. Socket.IO es probado en la industria para chats escalables; de hecho Slack y Discord usan websockets similares para lograr actualizaciones instantáneas. Cada cliente de Slack mantiene una conexión WebSocket persistente para recibir eventos en tiempo real [46], y nosotros haremos lo mismo. Usar Socket.IO nos simplifica manejar reconexiones automáticas, caídas, etc., ya que la librería integra *fallbacks* y reconexión exponencial. Además, el backend ya tiene eventos definidos como `new-message`, `typing`, `user-online`, etc. [11] [13] que el front deberá escuchar para actualizar la UI inmediatamente (nuevo mensaje entrante, indicadores de quién está escribiendo, etc.). *Nota:* Consideramos alternativas como **Ably** o **Pusher** (servicios externos de tiempo real), pero dado que tenemos servidor propio optimizado (capaz incluso de exceder estándares de Slack/WhatsApp [47]), preferimos control total con Socket.IO. Esto evita dependencias de terceros y nos permite aprovechar funciones custom (p. ej. *rate limiting* por evento ya implementado [16]).
- [ ] **Gestión de notificaciones**: Se implementará en dos niveles:
- **Notificaciones in-app**: para feedback rápido al usuario. Usaremos un componente de **Toast** (posiblemente aprovechando Shadcn UI que trae uno) para mostrar mensajes efímeros (ej: "Mensaje enviado " o errores). Esto mejora la UX sin molestar con alerts.
- **Notificaciones Push (Sistema)**: dado que queremos una experiencia tipo Slack, integraremos **Web Push Notifications** vía **Service Workers**. Esto permitirá que, si el usuario tiene la PWA instalada o el sitio abierto en segundo plano, pueda recibir avisos de nuevos mensajes incluso con la app inactiva [48]. Aprovecharemos que ya usamos Firebase en backend: podemos utilizar **Firebase Cloud Messaging (FCM)** para simplificar el envío de push a webs y potencialmente apps móviles. El plan es registrar desde el front la suscripción push del navegador (Service Worker) y almacenarla, para que el backend envíe notificaciones via FCM o WebPush API cuando lleguen mensajes a un usuario. Esto **reengagea** al usuario como lo haría una app nativa [49] [48]. También evaluaremos usar la API Notifications del navegador con el Service Worker escuchando eventos push [50]. *Nota:* Apple iOS recientemente empezó a soportar web push (con restricciones), así que la idea es cubrir al menos Chrome/Firefox/Edge. En resumen, la app será **PWA** (con manifest y SW) para aprovechar estas capacidades.
- [ ] **Herramientas de desarrollo**:
- **Linters y formateo**: Configuraremos **ESLint** (con reglas recomendadas para Svelte/TS) y **Prettier** para formato consistente. Así se asegura código limpio; incluso podemos usar **Husky** + **lint-staged** para correr linter en *pre-commit*. Esto previene que código mal formateado o con errores simples llegue al repo.
- **Commits**: Podemos adoptar Conventional Commits o al menos buenos mensajes, y quizá configurar **Commitlint** si se ve valor (menor prioridad).
- **Testing**: (Se detalla en sección 7, pero mencionar) Añadiremos frameworks como **Vitest** (rápido para TS, compatible con SvelteKit) o **Jest** para pruebas unitarias desde el inicio.

Toda elección de tecnología será documentada en un archivo de decisiones (`docs/decisions.md`) para futura referencia, incluyendo por qué se eligió sobre otras. Esto ayudará a *onboard* nuevos desarrolladores y justificar el stack ante stakeholders.

3. Arquitectura de carpetas y patrones

- [ ] **Estructura base de proyecto**: Organizaremos el código de forma modular y escalable. Una propuesta inicial de estructura (ajustada a SvelteKit) es:

```
frontend/
├── src/
│   ├── routes/          # Pages y endpoints SvelteKit (equivale a
páginas vistas)
│   │   ├── +layout.svelte    # Layout principal (nav lateral, etc.)
│   │   ├── +page.svelte      # Página inicial / dashboard
│   │   ├── login/
│   │   │   └── +page.svelte  # Página de login
│   │   ├── conversations/
│   │   │   ├── +page.svelte          # Lista de conversaciones
│   │   │   └── [id]/
│   │   │       └── +page.svelte      # Página de conversación
individual (chat)
│   │   └── profile/
│   │       └── +page.svelte   # Página de perfil de usuario
│   ├── lib/
│   │   ├── components/   # Componentes reutilizables (Button.svelte,
MessageItem.svelte, etc.)
│   │   ├── stores/       # Stores Svelte (estado global: usuario, etc.)
│   │   ├── hooks/        # Hooks o funciones utilitarias (por ej,
useAuth() si fuese React; en Svelte, composables)
│   │   ├── utils/        # Utilidades generales (formateo fecha,
cliente Axios config)
│   │   ├── services/     # Lógica de acceso a APIs (e.g.,
AuthService.ts con funciones login/refresh)
│   │   └── types/        # Definiciones de tipos/interfaces TS
│   ├── styles/           # CSS global o configuración Tailwind (ej:
tailwind.config.js)
│   └── app.d.ts          # Definiciones TS globales (SvelteKit)
├── static/               # Archivos estáticos (iconos, imágenes,
manifest.json para PWA)
├── tests/                # Pruebas unitarias/integración
├── package.json
└── vite.config.js        # Configuración de bundler (SvelteKit usa
Vite)
```

> *Nota:* De ser React/Next, la estructura sería similar pero con `/pages` o `/app` en lugar de `routes/`, y quizá una carpeta `/contexts` en lugar de stores. Lo importante es separar **concerns**: componentes presentacionales vs lógica de negocio vs utilidades.

- [ ] **Módulos por funcionalidad**: Vamos a aplicar una organización modular: los componentes y stores se agruparán por dominio cuando tenga sentido. Ejemplo:
- Módulo **auth**: componentes de Login, store de auth (usuario y tokens), servicio AuthService.
- Módulo **chat**: componentes de ConversationList, ChatWindow, MensajeItem, stores relacionados (ej. store de conversaciones en memoria, etc.), servicio ChatService (llamadas a `/messages` y `/conversations`).

Esto ayuda a la escalabilidad: a medida que agreguemos campañas, chatbot, etc., cada módulo tiene su espacio. Documentaremos esta arquitectura en `docs/ARCHITECTURE.md` como sugieres, para que cualquier desarrollador nuevo entienda la estructura rápidamente. * [ ] **Patrones de diseño de componentes**: - Usaremos componentes **presentacionales** (solo UI, props) vs **contenedores** (manejan datos) donde aplique. En Svelte, probablemente tengamos componentes que reciben datos ya preparados para mostrar (presentacionales), mientras la lógica de fetch estará en *load functions* o en el componente padre contenedor. - Aplicaremos patrón de **composición** en vez de herencia: componentes pequeños reutilizables (ej: <MessageItem>, <MessageList>, <ChatInput>) que se componen en vistas mayores. - **Custom Hooks/Composables**: Encapsularemos lógica repetitiva en funciones reutilizables. Por ej, un hook `useSocket()` para conectar al socket y exponer eventos, un hook `useNotifications()` para solicitar permiso de notificación y manejar suscripciones, etc. Esto mantiene los componentes limpios. - **Manejo de estado**: Centralizado en stores para datos globales; evitando *prop drilling*. For local state dentro de un componente, Svelte reactividad local es suficiente (no saturar el store global con todo). - **CSS design system**: Con Tailwind definiremos en config colores, fuentes y spacings acorde a la identidad (si existe guía de estilo). Posiblemente crearemos componentes base estilizados (ej: <Button variant="primary">) para un diseño consistente. * [ ] **Documentación de arquitectura**: Se elaborará un `ARCHITECTURE.md` que explique esta estructura, los principios de patrones usados y convenciones (nomenclatura de archivos, etc.). También incluiremos diagramas simples si ayuda (por ejemplo, un diagrama de componentes principales y cómo interactúan con stores y servicios).

En resumen, la arquitectura será modular, con separación clara de responsabilidades, facilitando la colaboración en equipo y evitando un monolito desordenado. Cada pieza del frontend tendrá su lugar definido.

4. Configuración inicial del repositorio

Antes de codificar funcionalidades, dejaremos listo el entorno de trabajo y la integración continua:

- [ ] **Inicializar repositorio Git**: Crear el repositorio (`git init`) y hacer un primer commit base. Añadir un remoto (GitHub, GitLab) según corresponda al proyecto.
- [ ] **.gitignore**: Configurar `.gitignore` para excluir `node_modules`, archivos de build (`.svelte-kit/`, `build/`), `.env` y credenciales, y cualquier otro artefacto (ej: coverage, logs). Usaremos plantillas estándar para Node/Svelte.
- [ ] **Instalar dependencias base**: SvelteKit, Tailwind, etc. Configurar Tailwind (generar config) e integrar con SvelteKit (postcss). Instalar Axios, Socket.IO client, etc. Este paso realmente ocurre después de planificar, pero conviene enumerar las dependencias clave en la documentación del repo.
- [ ] **Configurar linters y formateo**: Añadir ESLint con la configuración recomendada de Svelte (`eslint-plugin-svelte`), incluir reglas de TypeScript y posiblemente Prettier. Incluir un script npm `"lint"` que revise todo. Lo mismo con Prettier (archivo de config). Ejecutar una pasada inicial para asegurarnos que todo el repo sigue el estilo desde el inicio.

- [ ] **Pre-commits hooks**: Usar **Husky** para configurar un gancho pre-commit que ejecute el linter y quizás los tests. Así no entra código que rompe la build o viola estilos. Por ejemplo, `"husky": { "hooks": { "pre-commit": "npm run lint && npm run test" } }`.
- [ ] **CI/CD**:
- Si desplegaremos en **Vercel**, aprovecharemos su CI integrada: cada push a main (o a ramas específicas) puede disparar deploy previo a producción. Configuraremos el proyecto en Vercel vinculando el repo. Vercel es ideal si hubiéramos usado Next, pero soporta SvelteKit también mediante adaptador estático o serverless.
- Alternativamente, podemos usar **Railway** o **Netlify** para desplegar la app. Railway se mencionó para backend, pero para frontend preferimos Vercel por su foco en front.
- **GitHub Actions**: Implementaremos un workflow de CI que al hacer PR ejecute build y tests, asegurando que las contribuciones no rompan nada. En producción, podríamos requerir que CI pase antes de hacer merge.
- Documentar en `docs/DEPLOYMENT.md` cómo es el flujo de CI/CD (por ejemplo: branch `develop` se deploya a un entorno *staging*, branch `main` a producción).
- [ ] **README.md**: Actualizar el README del repo con:
- Descripción del proyecto, lista de tecnologías.
- Pasos de instalación y ejecución (ej: `npm install`, `npm run dev`).
- Cómo correr linters/test.
- Enlaces a documentación relevante (arquitectura, etc.). Esto asegurará que cualquier desarrollador pueda iniciar rápidamente.
- [ ] **Variables de entorno**: Crear un archivo `.env.example` con las variables necesarias en el front. Por ejemplo:
- `VITE_API_URL` (endpoint base del backend, ej. `http://localhost:3001`).
- `VITE_WS_URL` (URL del Socket.IO, ej. `ws://localhost:3001`).
- Keys de servicios externos si hubiera (ej: para FCM, una `VITE_FIREBASE_PUSH_KEY` o algo así, dependiendo implementación).

En SvelteKit, variables que empiezan con `VITE_` son expuestas al front. Dejaremos claro en la doc cuáles hay que configurar. **Importante:** no exponer secretos. La mayoría de config del front no son sensibles (URLs, etc.). Si necesitáramos usar el SDK de Firebase (por ej. para FCM), sus keys públicas podrían estar aquí también. * [ ] **Commit inicial limpio**: Haremos un commit inicial conteniendo la estructura de carpetas, configs, y quizás un "Hello World" mínimo (ej: página principal con mensaje). **Sin lógica de negocio aún**. Este commit servirá para comprobar que el pipeline CI/CD, el deploy, etc., funcionan en vacío. A partir de ahí, iremos construyendo módulo por módulo.

Resultado esperado: un repo listo para trabajar cómodamente, con guardrails de calidad (lint/test), con documentación básica, y con deployment continuo funcionando. Esto sienta bases sólidas para desarrollo ágil sin "sorpresas" después.

5. Integraciones críticas antes de empezar

Antes de implementar las funcionalidades finales, debemos verificar y preparar la comunicación con el backend y otros servicios críticos:

- [ ] **Prueba de API REST (backend)**: Realizar una llamada de prueba al backend para asegurarnos de la conectividad. Por ejemplo, usar *fetch* o Axios desde la consola o una pequeña función para llamar al *health check* (`GET /health`) del backend [51]. Si responde `{"status":"healthy", ...}` [52], confirmamos que la URL base y CORS están correctos. Luego, probar el flujo principal:

- **Login**: llamar `POST /api/auth/login` con credenciales de prueba (el backend envía en su README un ejemplo con `admin@utalk.com/password123` [53]). Verificar que obtenemos `200 OK` con un `accessToken` y datos de usuario [4] [54]. Almacenar ese token para siguientes pruebas.
- **Obtener conversaciones**: con el token, llamar `GET /api/conversations` (lista de conversaciones). Debería devolver 200 con una lista paginada [55] [56]. Confirmar estructura: cada conversación tiene `id`, `contactName`, `lastMessageAt`, etc.
- **Obtener mensajes**: probar `GET /api/conversations/{id}/messages` para una conversación concreta (si el backend lo soporta) o `GET /api/messages?conversationId=...`. Debería traer mensajes con sus campos (content, timestamp, status) [57] [58].
- **Enviar mensaje**: probar `POST /api/messages` con un cuerpo `{"conversationId":"conv_123", "content":"Hola", "type":"text"}` [59] y el token auth. Debería retornar 201 con datos del nuevo mensaje (id, content, status "sent") [60]. **Importante**: verificar si el nuevo mensaje aparece vía socket también (ver siguiente punto).

Estas pruebas iniciales garantizarán que la API está lista y que conocemos sus respuestas. Además, sirven para diseñar nuestros modelos front (interfaces TS para Conversation, Message, etc.). * [ ] **Prueba de conexión en tiempo real**: Configurar temporalmente el **cliente Socket.IO** (podemos escribir un pequeño script o usar la consola del navegador): - Con el token JWT obtenido, intentar conectar a `ws://localhost:3001` (o la URL correspondiente) usando `io(URL, { auth: { token } })`. El backend espera el token en el handshake [44]. Si todo va bien, el socket `.on("connect")` debe dispararse. Si hay error de auth, veremos un `connect_error` (lo manejaremos en código para por ejemplo redirigir a login si token inválido). - Suscribir a un evento de prueba: por ejemplo, el backend podría emitir `conversation-joined` tras uno unirse a una sala. O más simple: emitir desde cliente un evento `join-conversation` con un ID válido, y escuchar `conversation-joined` [61]. Si recibimos respuesta, sabemos que la unión a salas funciona. - Probar envío de mensaje via socket: El flujo previsto es: tras hacer POST /messages, el backend emitirá un evento `new-message` a todos en esa conversación [11]. Entonces, del lado cliente, simular enviar mensaje vía REST y ver si nuestro listener de `new-message` capta algo. Alternativamente (si backend lo permite), usar socket.emit('new-message', {...}) directamente. Pero según arquitectura, **el backend probablemente requiere usar el endpoint REST para crear mensaje**, y luego internamente emite `new-message` a sockets como notificación [6]. Confirmaremos esto: al hacer POST de antes, deberíamos recibir un evento `new-message` con los datos del mensaje creado. - Probar otros eventos: Por ejemplo emitir desde consola `socket.emit('typing', { conversationId: X })` y ver si backend responde algo (puede que no directamente). Pero al menos podemos observar en network si el ping-pong de WS está activo.

Todo esto nos asegura que la integración en tiempo real está funcional y lista para ser usada en nuestro código. El backend UTalk ya implementa autenticación robusta en socket (JWT obligatorio, etc. [44]), así que nuestro front deberá siempre conectar después de login exitoso (guardando token). * [ ] **Documentar endpoints críticos**: A medida que probamos, recopilamos los detalles para el equipo: - URL base (ej: `https://api.utalk.com` en prod, `http://localhost:3001` en dev). - Endpoints usados en MVP: **Auth** (`/api/auth/login`, `/auth/refresh`, `/auth/logout`), **Conversations** (`GET /api/conversations`, `/conversations/:id`), **Messages** (`GET /api/messages?conversationId=`, `POST /api/messages` para enviar, `PUT /api/conversations/:id/messages/:msgId/read` para marcar leído si existe). - Para cada uno, qué datos envía y retorna (esto en buena parte está en `docs/API.md` del backend).

Esta documentación interna ayudará a quienes implementen los servicios front a no tener que leer todo el backend. Podríamos resumirlo en un confluence o en markdown en `/docs/API_FRONTEND_USAGE.md`. * [ ] **Configurar manejo de errores global**: Decidir cómo manejaremos expiración de sesión, errores 401, etc. Por ejemplo, dado que tenemos refresh token, debemos integrar en Axios un interceptor: si 401 y la respuesta indica token expirado, llamar refresh endpoint y reintentar la original. Estas mecánicas deben planearse ahora. Posiblemente escribamos un **AuthService.refreshToken()** que se active automáticamente. También planificar manejo de errores global: un componente `<ErrorBoundary>` o en SvelteKit aprovechar la forma de manejar errores (crear +error.svelte para páginas de error global). * [ ] **Servicios externos**: Verificar si el front necesita integrarse directamente con alguno. Por ejemplo: - **Firebase**: en principio, el front no usará directamente Firestore ni Auth, todo pasa por backend. **Excepción**: si implementamos notificaciones push via FCM, necesitaremos incluir el SDK de Firebase Messaging en el front para registrar el token de dispositivo. Deberíamos probar eso: inicializar Firebase app con config (non-sensitive config de proyecto) y pedir permiso de notificaciones. Esto se puede probar manualmente y documentar los pasos (y asegurar que Apple/Android PWA son considerados). - **Mapas, otros**: No aplica mucho en chat.

Por ahora, parece que el front se comunica solo con **UTalk API** y con **Socket.io**, lo cual simplifica. * [ ] **Revisar CORS y seguridad**: Asegurar que el backend permite nuestras peticiones (CORS configurado para nuestro origen dev). En el `.env` backend vimos `CORS_ORIGINS` [62]. Nos aseguraremos de incluir nuestro dominio front al desplegar (ej: `http://localhost:5173` en dev, y dominio real en prod). Esto lo coordinaremos con backend config. * [ ] **Verificar versiones**: Confirmar que usamos versiones compatibles: p. ej Socket.IO client v4 para server v4, Axios v1, etc. También que Node versión para dev es la correcta (Node 18+). * [ ] **Performance baseline**: Opcional, pero podríamos desde inicio configurar profiling sencillo. SvelteKit con Vite permite analizador de bundle. Podemos hacer un build inicial y ver tamaño (debe ser muy pequeño con solo scaffolding). Esto de base nos servirá para monitorear que al añadir dependencias no se dispare el bundle (recordemos que Slack-like apps pueden crecer, pero queremos mantenerlo optimizado).

En síntesis, este paso nos da **confianza** de que cuando construyamos el front, la comunicación con el back no será un escollo. Identificamos pronto cualquier problema de entorno o credenciales, en vez de al final.

6. Planificación de versiones y avance

Para organizar el desarrollo, definimos hitos (*milestones*) y un plan incremental:

- [ ] **Versión MVP (v0.1) – Autenticación + Chat básico**:
- **Objetivo**: Tener un sistema funcional donde un agente pueda iniciar sesión y enviar/recibir mensajes en tiempo real en una interfaz mínima.
- **Incluye**: Login/logout, pantalla de lista de conversaciones (puede ser sencilla, ej. lista de conversaciones activas del usuario), pantalla de chat mostrando mensajes, envío de texto, recepción de mensajes entrantes (socket). Indicadores básicos: quizá un marcador de "en línea" para contactos si fácil.
- **Criterio de listo**: Un usuario agente puede conversar con otro (o con un cliente simulado) en tiempo real. La app maneja expiración de sesión (token refresh). UI sencilla pero sin errores. Código cubierto con pruebas unitarias de utils y quizás un test de integración simulado (podemos simular una conexión WS).
- **Tiempo estimado**: ~4 semanas.
- [ ] **Versión 1.0 – Chat completo + Perfil**:
- **Objetivo**: Front listo para producción con todas features core de chat interno.

- **Incluye**: Todo MVP, más: interfaz mejorada (diseño pulido con Shadcn components), soporte de **mensajería enriquecida** (enviar imágenes/archivos adjuntos), indicador "Usuario está escribiendo...", **marcar mensajes como leídos** (con confirmación visual de ticks, etc.), vista de **perfil de usuario** (y permitir cambiar nombre/contraseña), notificaciones push integradas.
- **Roles**: Soporte pleno de roles admin/agent: por ejemplo un admin podría tener un panel dashboard (métricas) y gestión de usuarios – esto último podría ser pospuesto si no crítico.
- **Criterio de listo**: La aplicación replica una experiencia tipo Slack básica: multi-conversación, multi-usuario, reliable. Pases de QA completos, sin bugs críticos.
- **Estimado**: +4-6 semanas tras MVP.
- [ ] **Versión 1.1 – Características multicanal & extra**:
- **Campañas**: interfaz para que usuario cree campañas de mensajería masiva (apoyándose en endpoints `/api/campaigns` [63]). Esto incluiría formularios avanzados (p. ej. seleccionar audiencia).
- **Bot/IA**: UI para interactuar con el chatbot (quizá una conversación especial tipo “Chatbot”).
- **Base de conocimientos**: sección para buscar artículos (según el backend, existe KB).
- **Mejoras de UX**: por ejemplo, *drag & drop* de archivos al chat, reacciones a mensajes, threads (responder a mensajes, backend ya soporta `replyToMessageId` [64]).
- **Criterio**: Expansión funcional terminada, app lista para casos de uso más amplios.

(Las versiones 1.1 en adelante pueden planificarse con metodologías ágiles, ajustando en backlog). * [ ] **ROADMAP y seguimiento**: - Crearemos un documento `docs/ROADMAP.md` con estos hitos, fechas tentativas y prioridades. Será actualizado conforme avanza el proyecto. - Listaremos posibles **riesgos** y mitigaciones: - *Riesgo:* El equipo no domina SvelteKit (mitigar con capacitación inicial, pair programming al inicio). - *Riesgo:* Integración socket compleja (mitigar con pruebas en entorno controlado primero, como ya planificado). - *Riesgo:* Performance de listas largas de mensajes (mitigar usando técnicas virtual scroll en caso de miles de msgs, y pruebas de carga). - *Riesgo:* Requerimientos cambiantes (mitigar trabajando con metodología ágil – sprints cortos y demos frecuentes al Product Owner para realinear). - **Prioridades semanales**: Usaremos probablemente sprints de 1-2 semanas. Al inicio de cada sprint definiremos tareas concretas (e.g., Sprint 1: estructura + login básico; Sprint 2: pantalla chat con list/scroll; Sprint 3: realtime integration; etc.). - **Control de avance**: Setup de **tracking de tareas** (Jira, Trello o GitHub Projects) donde cada elemento de esta checklist quizás sea un *story*. Esto permitirá marcar cada [ ] como [x] cuando esté listo.

- [ ] **Entrega y demo**: Al final de MVP y 1.0 haremos demos funcionales al equipo stakeholder para feedback. También pruebas de usuario internas para pulir UX.

Este plan de versiones asegura que primero logramos un núcleo funcional, luego iteramos agregando lo demás. Así obtenemos valor utilizable rápidamente (MVP) y reducimos riesgo, en vez de intentar el *big bang* al final.

7. Checklist de calidad continua

La calidad se garantizará desde el día 1 mediante prácticas integradas en el flujo de desarrollo:

- [ ] **Testing automatizado desde el inicio**:
- **Pruebas unitarias**: Implementaremos tests unitarios para funciones puras y componentes críticos. Por ejemplo, utilidades de formateo (fechas de mensaje), stores (reducers o funciones), y componentes de presentación (usar Svelte Testing Library o similar para verificar que muestran datos correctamente dados props). Al menos cubrir casos base: login form validation, mensaje componente corta texto largo adecuadamente, etc.

- **Pruebas de integración**: A medida que tengamos módulos completos, escribir tests que simulen escenarios completos. Por ejemplo, con un *mock* del servidor (o utilizando un entorno de staging del backend), probar el flujo login + abrir conversación + enviar mensaje y comprobar que el estado global se actualiza. Podemos usar herramientas como Cypress o Playwright para pruebas end-to-end en el navegador (por ejemplo, levantar app en modo dev, simular usuario escribiendo y ver resultado en DOM).
- **Cobertura**: Apuntamos a un % decente (ej. >80% en utils y lógica). Sin obsesión, pero útil.
- [ ] **Revisiones de código (code review)**: Todo *pull request* deberá ser revisado por al menos un colega antes de merge. Esto asegura estándares de calidad, detección de bugs temprana y difusión de conocimiento en el equipo. Usaremos Github PRs con checklist (¿pasa CI? ¿sigue guías estilo? ¿hay pruebas? etc.).
- [ ] **Documentación continua**:
- Cada componente complejo tendrá comentarios o README contextual. Ej: un componente ChatWindow.svelte podría llevar en comentarios la descripción de cómo funciona la paginación de mensajes o cómo se suscribe a sockets.
- Usar JSDoc/TSdoc en funciones utilitarias para que otros desarrolladores entiendan su propósito rápidamente.
- Mantener actualizado el `/docs` del repo: guía de despliegue, de seguridad, etc., tomando inspiración del backend que ya tiene docs (Deployment, Security, Testing) [65] [66] .
- [ ] **Linting & Formatting estrictos**: Gracias a ESLint/Prettier integrados, todos los commits tendrán formato consistente. Configuraremos reglas para código accesible (eslint-plugin-jsx-a11y si fuera React, en Svelte hay linter para accesibilidad también). No se permitirán warnings de linter en build final.
- [ ] **Performance monitoring**: Aunque es front, podemos integrar herramientas como **Lighthouse CI** en el pipeline para vigilar performance (p.ej. que Time to Interactive se mantenga bajo cierto umbral). Y en producción, podríamos usar **Sentry** o similares para capturar errores runtime no detectados en test.
- [ ] **Seguridad front**: Seguir buenas prácticas: nunca exponer info sensible en código, sanitizar input (la mayoría ya lo hace backend con Joi [67] , pero en front validaremos formularios para usabilidad). Considerar usar `Content Security Policy` apropiada (el backend ya usa Helmet [68] , pero en front al ser SPA se complementa configurando meta CSP si hace falta). También evitar vulnerabilidades XSS en componentes (Svelte escapa contenido por defecto, excepto donde marquemos HTML puro).
- [ ] **Retroalimentación de usuarios**: Una vez en testing interno o beta, recopilar feedback. Podríamos instrumentar analíticas simples (ej: cuántos mensajes envía un usuario al día) para detectar uso. Pero como hay un dashboard analytics en back [69] , quizá solo enfocarnos en UX feedback directo.
- [ ] **Proceso de despliegue controlado**: En producción, usar despliegues *canary* si posible (por ejemplo, Vercel supports deploying preview to a % of users). Al menos, hacer staging testing antes de pasar a prod.
- [ ] **Mantenimiento**: Definir responsables del mantenimiento post-release, asegurar que hay rotación de guardia en caso de bugs urgentes (particularmente porque es un chat, se espera alta disponibilidad).

En esencia, trataremos el código front con el mismo nivel de rigurosidad que el backend enterprise. Testing y calidad no son opcionales: así como el backend fue auditado y probado [70] , el front debe estar a la altura, evitando regresiones y garantizando una experiencia sólida a los usuarios finales.

Resumen: Con esta planificación, disponemos de un **mapa detallado** para construir un frontend de chat de clase mundial, combinando lo mejor de la tecnología actual (SvelteKit, Tailwind, Socket.IO, etc.)

con prácticas de ingeniería robustas (testing continuo, arquitectura modular). Siguiendo este documento paso a paso, nuestro equipo podrá desarrollar un frontend **“a la altura”** del backend o incluso superior, logrando una aplicación tipo Slack moderna, escalable y preparada para el futuro. ¡Manos a la obra! [47] [46]

---

1 2 8 15 51 52 53 62 65 66 67 68 69 70 README.md
https://github.com/isaavedra43/Utalk-backend/blob/a9d5f8df958eb0dbfc52fe36b07a2497d9002835/README.md

3 4 5 7 9 10 54 55 56 57 58 59 60 63 API.md
https://github.com/isaavedra43/Utalk-backend/blob/a9d5f8df958eb0dbfc52fe36b07a2497d9002835/docs/API.md

6 14 64 MessageController.js
https://github.com/isaavedra43/Utalk-backend/blob/a9d5f8df958eb0dbfc52fe36b07a2497d9002835/src/controllers/MessageController.js

11 12 13 16 17 42 43 61 enterpriseSocketManager.js
https://github.com/isaavedra43/Utalk-backend/blob/a9d5f8df958eb0dbfc52fe36b07a2497d9002835/src/socket/enterpriseSocketManager.js

18 Svelte vs Next.js: Which JavaScript Framework Should You Choose?
https://www.dhiwise.com/post/svelte-vs-nextjs

19 Which Front-End Framework Feels Most "Future-Proof" in 2025? - Community - SitePoint Forums | Web Development & Design Community
https://www.sitepoint.com/community/t/which-front-end-framework-feels-most-future-proof-in-2025/475770

20 React vs Vue vs Svelte: Choosing the Right Framework for 2025
https://medium.com/@ignatovich.dm/react-vs-vue-vs-svelte-choosing-the-right-framework-for-2025-4f4bb9da35b4

21 22 23 Why Front-End Developers Should Use TypeScript in 2025 - DEV Community
https://dev.to/priya_khanna_44234bba65fb/why-front-end-developers-should-use-typescript-in-2025-2m26

24 25 26 27 Top 7 CSS Frameworks in 2025
https://www.wearedevelopers.com/en/magazine/362/best-css-frameworks

28 Material UI vs. ShadCN UI - Which Should You be using in 2024?
https://blog.openreplay.com/material-ui-vs-shadcn-ui/

29 Top 5 CSS Frameworks in 2024: Tailwind, Material-UI, Ant Design, Shadcn & Chakra UI
https://www.codingwallah.org/blog/2024-top-css-frameworks-tailwind-material-ui-ant-design-shadcn-chakra-ui/

30 31 32 33 34 35 State Management in 2025: Redux, Zustand, or React Query?
https://www.linkedin.com/pulse/state-management-2025-redux-zustand-react-query-sbtqc

36 37 38 39 React-Hook-Form vs Formik: The Good, Bad, and Ugly - Joyfill
https://joyfill.io/blog/react-hook-form-vs-formik-the-good-bad-and-ugly

40 Should I use fetch or axios to make API calls? : r/reactjs - Reddit
https://www.reddit.com/r/reactjs/comments/1hp5glg/should_i_use_fetch_or_axios_to_make_api_calls/

41 Say Goodbye To Axios In 2025 - JavaScript in Plain English
https://javascript.plainenglish.io/say-goodbye-to-axios-in-2025-04fc0772c01e

44 45 47 REALTIME_ARCHITECTURE.md
https://github.com/isaavedra43/Utalk-backend/blob/a9d5f8df958eb0dbfc52fe36b07a2497d9002835/REALTIME_ARCHITECTURE.md

46 Real-time Messaging - Engineering at Slack
https://slack.engineering/real-time-messaging/

48 49 50 Using Push Notifications in PWAs: The Complete Guide | MagicBell
https://www.magicbell.com/blog/using-push-notifications-in-pwas