ChatGPT

Plan Maestro para el Frontend de UTalk

Este documento presenta un plan detallado y paso a paso para construir desde cero el frontend de **UTalk**, enfocándose inicialmente en un sistema de **Login** robusto y un **Módulo de Chat** en tiempo real de clase mundial. La arquitectura propuesta prioriza escalabilidad, mantenibilidad y performance, preparando el camino para integrar posteriormente módulos como CRM, configuración (*settings*), campañas, base de conocimiento, etc. Se ha elegido un *stack* tecnológico moderno y eficiente, justificando cada decisión técnica. Además, se incluyen ejemplos de código claros y bien comentados (estilo *IA-friendly*) y una definición de hitos (*milestones*) con criterios **DONE** para cada fase del proyecto.

1. Justificación Técnica del Stack Seleccionado

Para un frontend de alto rendimiento y fácil escalabilidad, se propone utilizar **SvelteKit** junto con **Tailwind CSS** y la librería de componentes **shadcn-svelte**. A continuación, se detalla la justificación de esta elección tecnológica:

- **SvelteKit (Svelte)**: Svelte se destaca por su enfoque de *compilación* en tiempo de desarrollo, eliminando la sobrecarga del framework en el *runtime*. Esto da como resultado aplicaciones más rápidas y paquetes más pequeños en producción. A diferencia de frameworks tradicionales (React, Angular) que utilizan un *virtual DOM* y resuelven el estado en tiempo real, Svelte compila los componentes a código JavaScript altamente optimizado. Esta filosofía de "framework que desaparece mágicamente" implica que el *tooling* hace el trabajo pesado en build, liberando al navegador de ese costo [1]. En 2023 Svelte fue reconocido como el framework de UI más admirado en Stack Overflow, lo que refleja su madurez y excelente experiencia de desarrollador. **SvelteKit**, por su parte, es el *meta-framework* que proporciona enrutamiento, rendering del lado del servidor (SSR), generación estática y otras optimizaciones listas para usar. Esto nos da:
- *Server-Side Rendering* y *hydration* automática, mejorando SEO y tiempos de carga inicial.
- *Routing* basado en el sistema de archivos (convención `src/routes`), facilitando la escalabilidad modular por secciones (login, chat, CRM, etc).

- Integración nativa con TypeScript, manejo de entornos, y adaptabilidad a distintas plataformas de despliegue (Node, serverless, Edge, etc).

- **Tailwind CSS**: Como framework de utilidades CSS, Tailwind nos permite construir interfaces consistentes de forma rápida. Sus ventajas:

- Estilos altamente reutilizables mediante clases utilitarias, manteniendo el CSS mantenible sin crear cascadas complejas.
- Excelente performance gracias a generación de CSS purgado (solo las clases usadas) en build.
- Fácil implementación de diseño responsivo y temas (ej. modo oscuro) con utilidades predefinidas.

- Alinea con la filosofía de diseño de un sistema: se pueden definir colores de marca, fuentes y spacings en la configuración para asegurar un **diseño consistente** en toda la aplicación.

- **shadcn-svelte**: Es una colección comunitaria de componentes UI construidos con Svelte + Tailwind, portando los populares componentes de **shadcn/ui** (originalmente para React). Su

enfoque es inusual pero poderoso: en vez de ser una librería cerrada, proporciona el código fuente de los componentes para que el equipo lo copie, modifique y extienda según sus necesidades. Esto se traduce en un sistema de diseño totalmente bajo nuestro control, con defaults hermosos y accesibles. Algunas razones para usar shadcn-svelte:

- *Componentes pre-construidos y accesibles*: Incluye desde botones, diálogos, menús, hasta componentes complejos como tablas, calendarios y *drawers*. Todos con estilos consistentes basados en Tailwind, listos para adaptar a la identidad de UTalk.
- *Open Code*: Tenemos el código de cada componente, lo que permite personalización total sin *hacks*. No estamos limitados por la API de un paquete externo; si un componente necesita comportamiento especial, se modifica directamente el código fuente según nuestras necesidades [2].
- *Beautiful Defaults*: Los componentes vienen con estilos por defecto bien pensados, proporcionando un diseño atractivo *out-of-the-box*, acelerando la construcción de una UI de nivel profesional [3] [4].
- *AI-Ready*: Este enfoque de código abierto facilita que herramientas de IA (como Cursor AI) entiendan y aprendan del código base de nuestros componentes [5]. Dado que uno de los objetivos es tener ejemplos de código claros y autoexplicativos, shadcn-svelte encaja perfectamente con esa filosofía.

- En resumen, shadcn-svelte nos da una **biblioteca de componentes personalizable** que podemos integrar en nuestro proyecto SvelteKit, manteniendo consistencia visual y ahorrando tiempo al no reinventar componentes comunes, pero sin sacrificar la capacidad de modificarlos profundamente.

- **Otras tecnologías de soporte**: Además del trío principal, se considerarán:

- **TypeScript** en todo el proyecto para robustez y autocompletado. SvelteKit facilita su uso, previniendo muchos errores en desarrollo.
- **ESLint + Prettier** para asegurar un estilo de código consistente y atrapar problemas de calidad (incluyendo reglas de accesibilidad de Svelte).
- **Vitest** (o **Jest**) y **Testing Library (Svelte)** para pruebas unitarias y de componente, así como **Playwright** para pruebas end-to-end. Esto garantiza calidad desde el día 1.
- **Socket.io (Cliente)** para el módulo de chat en tiempo real, dado que el backend provee comunicación en tiempo real (suponiendo que el backend use Socket.io u otro protocolo WebSocket estándar). Socket.io facilita la reconexión automática, manejo de eventos y fallos de conexión, lo que es ideal para chat en tiempo real. Alternativamente, si el backend usa WebSockets puros, usaremos la API WebSocket nativa con una pequeña capa de reconexión manual.
- **Vite** (empacador que viene con SvelteKit) para compilación rápida y división de código (*code splitting*). Vite también nos permite **HMR** (Hot Module Reload) en desarrollo, mejorando la productividad.

¿Por qué no React/Angular/Vue? Si bien frameworks como React (con Next.js) o Vue (con Nuxt) podrían ser opciones viables, SvelteKit ofrece un equilibrio excepcional de simplicidad, rendimiento y características modernas. Su curva de aprendizaje es rápida, y produce menos código *boilerplate*. Dado que buscamos una base de código limpia, mantenible y óptima para futuros módulos, SvelteKit nos brinda esa solidez sin el peso de frameworks más complejos.

En conclusión, el stack **SvelteKit + Tailwind + shadcn-svelte** (potenciado por TypeScript y herramientas de calidad) es la elección ideal para UTalk. Ofrece *performance* superior, excelente experiencia de

desarrollador, una base de UI consistente y altamente personalizable, y se alinea con la visión de código abierto y claro que puede ser fácilmente comprendido y mejorado por desarrolladores y sistemas de inteligencia artificial.

2. Estructura Completa del Proyecto

A continuación se define la estructura de carpetas y módulos del proyecto frontend, organizada para favorecer la escalabilidad y el mantenimiento. La estructura sigue las convenciones de SvelteKit, con adaptaciones para nuestros dominios (auth, chat, etc.) y separación de responsabilidades clara:

```
utalk-frontend/
├── src/
│   ├── routes/
│   │   ├── +layout.svelte
│   │   ├── +layout.ts            # Carga inicial (ej. verificar auth global)
│   │   ├── +error.svelte         # Manejo global de errores (SSR)
│   │   ├── login/
│   │   │   ├── +page.svelte      # Página de Login
│   │   │   └── +page.server.ts   # Acciones de Login (autenticación
servidor)
│   │   ├── chat/
│   │   │   ├── +page.svelte      # Página principal del Chat (UI chat)
│   │   │   ├── +page.ts          # Carga de datos iniciales del chat (ej.
últimos mensajes)
│   │   │   └── socket.ts         # (Opcional) Endpoint para manejar sockets
si se requiere
│   │   ├── crm/                  # (Módulos futuros: CRM)
│   │   ├── settings/             # (Módulo futuro: Configuración)
│   │   ├── campaigns/            # (Módulo futuro: Campañas)
│   │   └── knowledge-base/       # (Módulo futuro: Base de conocimiento)
│   ├── lib/                      # Código compartido (utils, stores,
componentes, servicios)
│   │   ├── components/           # Componentes reutilizables (UI)
│   │   │   ├── ui/               # Componentes atómicos (botones, inputs,
modales) posiblemente de shadcn-svelte
│   │   │   ├── chat/             # Componentes específicos del chat
(MessageList, MessageItem, ChatInput, etc.)
│   │   │   └── ...               # Otros componentes modulares por dominio
│   │   ├── stores/               # Almacenes de estado global (Svelte
stores)
│   │   │   ├── auth.store.ts     # Store para datos de usuario autenticado
(session)
│   │   │   ├── chat.store.ts     # Store para mensajes de chat, estados de
conexión
│   │   │   └── ...               # (posibles stores para otros módulos
futuros)
│   │   ├── services/             # Lógica de comunicación con backend
│   │   │   ├── api.ts            # Cliente API REST (fetch) configurado con
base URL, headers
```

```
│   │   │   ├── auth.service.ts   # Servicios de autenticación (login,
logout, refresh token)
│   │   │   ├── chat.service.ts   # Servicios de chat (ej. inicializar
socket, enviar mensaje, manejar reconexión)
│   │   │   └── file.service.ts   # Servicio para subida de archivos (upload
a servidor o S3)
│   │   ├── utils/                # Utilidades generales (formateo de fechas,
manejo de errores, etc.)
│   │   ├── types/                # Definiciones de tipos/interfaces
TypeScript usadas en el app (Message, User, etc.)
│   │   └── constants.ts         # Constantes globales (ej. nombres de
storage, configuraciones)
│   ├── app.css                   # CSS global (incluye Tailwind directives)
│   ├── app.html                  # Plantilla HTML (personalizable si se
requiere meta tags globales)
│   └── hooks.server.ts           # Hooks de SvelteKit (manejo global de
autenticación por request, etc.)
├── static/                       # Archivos estáticos (imágenes públicas,
iconos, manifest PWA, etc.)
├── tests/
│   ├── unit/                     # Pruebas unitarias (.spec.ts) para
servicios, stores, utilidades
│   ├── component/                # Pruebas de componentes Svelte aislados
(con Testing Library)
│   └── e2e/                      # Pruebas End-to-End (Playwright/Cypress)
├── scripts/                      # Scripts utilitarios (ej: scripts de
despliegue, generación de datos de prueba)
├── package.json
├── svelte.config.js              # Configuración de SvelteKit (incl.
adaptador, preprocess Tailwind)
├── tailwind.config.js            # Configuración de Tailwind (colores,
fuentes, plugins)
├── vite.config.ts                # Configuración de Vite (si se necesita
custom)
├── .eslintrc.cjs                 # Configuración de ESLint
├── .prettierrc                   # Configuración de Prettier
├── .husky/                       # Hooks de git (husky) para CI local
└── ci/                           # Configuración de CI/CD (pipeline,
workflows)
```

Detalles clave de la estructura:

- `src/routes`: sigue la convención de SvelteKit para definir páginas y endpoints. Cada sección (login, chat, etc.) tiene su subcarpeta con los archivos +page (componentes de página) y archivos .server o .ts para la lógica de carga de datos o acciones.
- Hemos previsto rutas vacías para módulos futuros: **crm**, **settings**, **campaigns**, **knowledge-base**, de modo que el routing está preparado. Por ejemplo, `/crm` tendrá su propia página cuando se implemente, bajo `src/routes/crm/+page.svelte`. Esto asegura que la navegación y estructura crezcan orgánicamente sin refactorizaciones mayores.

- Un **+layout.svelte** global en `src/routes` nos servirá para definir el armazón común (por ejemplo, un layout con barra de navegación lateral si el usuario está logueado, o layout básico público para login). Podemos usar layouts anidados si, por ejemplo, las secciones internas comparten un marco común distinto al de la pantalla de login.

- Un **+error.svelte** global capturará errores no manejados en la aplicación, mostrando una UI de error amigable.

- `src/lib/components`: aquí viven los componentes reutilizables. Se subdividen en:

- **ui/**: componentes básicos y atómicos (muchos vendrán de shadcn-svelte). Ej: `<Button>`, `<Input>`, `<Modal>`, `<Avatar>` etc. Estos probablemente se agreguen copiando del repositorio de shadcn-svelte y adaptando a nuestro proyecto (p.ej., usando nuestra configuración de Tailwind para theme).
- **chat/** (u otros dominios): componentes específicos de funcionalidad. Ej: `ChatMessage.svelte` (un mensaje individual), `ChatList.svelte` (lista de conversaciones), `ChatWindow.svelte`, `FileUploader.svelte` (componente para adjuntar archivos en chat), etc. Aunque son particulares a chat, los mantenemos modulares por si se reutilizan en otros contextos (por ejemplo, un componente `Avatar` es general, pero uno `ChatMessage` quizás solo aplique en chat).

- Podemos añadir subcarpetas en components para CRM, settings, etc. cuando se desarrollen, manteniendo cada módulo aislado en cuanto a componentes de UI específicos.

- `src/lib/stores`: almacenes de estado global usando Svelte stores. Aquí definiremos:

- Un **auth.store.ts** con, por ejemplo, un `userStore` para información del usuario autenticado (p. ej., datos del perfil, token JWT si se maneja en memoria, estado de login).
- Un **chat.store.ts** para manejar el estado de chat: lista de mensajes actuales, usuarios conectados, estado de conexión del socket (conectado/intentando reconectar), etc.

- La idea es centralizar estados compartidos en stores Svelte (utilizando `writable` / `readable`), en lugar de prop-drilling. Stores facilitan reactividad global; cualquier componente suscrito se actualiza cuando cambia el estado. Por ejemplo, el `userStore` puede ser usado tanto en la barra de navegación (para mostrar nombre del usuario) como en la lógica de rutas protegidas.

- `src/lib/services`: funciones o clases que encapsulan llamadas al backend y lógica empresarial del frontend:

- **api.ts** podría configurar el cliente fetch o axios con la URL base de la API (por ejemplo, sacada de variable de entorno), y quizás una función genérica `apiFetch(endpoint, options)` que incluya el token de auth automáticamente en headers o maneje errores globales.
- **auth.service.ts**: maneja la interacción con autenticación. Por ejemplo, funciones `login(credentials)` que hagan `POST /auth/login` al backend, almacenen token/usuario, y configuren el estado del cliente (p.ej., cookie o store). También `logout()` para limpiar sesión, y `refreshToken()` si aplica.
- **chat.service.ts**: se encarga de toda la comunicación de chat. Por ejemplo, inicializar el socket: `connectChatSocket()` usando Socket.io o WebSocket, suscribir a eventos de nuevos mensajes, envíos, notificaciones de usuario escribiendo, etc. También puede incluir métodos para obtener el historial (`fetchConversation(convId)` via REST, si el backend provee un endpoint HTTP para historial), o enviar mensajes `sendMessage(convId, content)`

internamente llamando al socket. La idea es separar esta lógica del componente Svelte para mantenerlo limpio.
- **file.service.ts**: abstracta la subida de archivos, por ejemplo usando el API REST (`POST /files` o similar) y quizás devolviendo una URL o ID que luego el chat utiliza. También podría manejar compresión o conversión de archivos antes de enviarlos, si fuera necesario.

- Cada servicio es independiente pero puede usar `api.ts` o compartirse utilidades. Esto ayuda a realizar pruebas unitarias más fácilmente (ya que podemos simular estos servicios sin tocar la UI directamente).

- `src/lib/utils`: utilidades generales. Ejemplos:

- `date.util.ts` con funciones para formatear fechas (última hora de mensaje, etc.).
- `validators.ts` con funciones de validación (ej., validar formato de email, contraseñas fuertes).
- `errors.ts` podría mapear códigos de error a mensajes de usuario (por ejemplo, error 401 -> "Credenciales incorrectas").
- `socket.utils.ts` con lógica de reconexión exponencial si usamos WebSocket nativo, etc. (si no se maneja en chat.service).

- `notifications.ts` para encapsular uso de la API de notificaciones del navegador (mostrar notificación de "Nuevo mensaje de X" cuando la app está en segundo plano).

- `src/lib/types`: definiremos interfaces y tipos TypeScript para las entidades principales, asegurando tipado consistente:

- Ej: `User`, `ChatMessage`, `Conversation`, `ChatEvent` (tipos de eventos de socket), etc., basados en la documentación del backend. Por ejemplo, si el backend `BACKEND_ADVANCED_LOGIC.md` describe la estructura de un mensaje (campos como id, texto, remitente, timestamp, estado de entrega, etc.), aquí crearemos esa interfaz TypeScript.

- Estos tipos se usan en stores y servicios para que el código sea auto-documentado. P. ej., `sendMessage(content: string, to: string): Promise<Message>` dejando claro que retorna un Message con los datos completos (id asignado por backend, timestamps, etc).

- Archivos de configuración y soporte:

- **svelte.config.js**: configurará el adaptador de despliegue (por ejemplo, el adaptador de Node.js para desplegar en nuestro servidor, o Vercel adapter si vamos a Vercel). También integrará `vitePreprocess()` con Tailwind y PostCSS, de modo que Svelte pueda procesar nuestras clases Tailwind y otros transformadores.
- **tailwind.config.js**: definirá el tema de diseño de UTalk: colores corporativos, fuentes, breakpoints, etc. Incluirá los presets necesarios para shadcn-svelte (por ejemplo, hay que incluir `tailwindcss-animate` si los componentes lo usan). Ajustaremos este archivo para que los componentes de shadcn-svelte luzcan acorde a la imagen de marca de UTalk.
- **.eslintrc.cjs**: incluirá configuraciones para Svelte (plugin `svelte`), TypeScript, y reglas de estilo. Se activarán reglas de accesibilidad (por ejemplo, asegurar que `<img>` tenga alt, que botones tengan type, etc.), así como reglas de código recomendadas por Svelte.
- **.husky/**: contendrá los ganchos pre-commit y pre-push. Por ejemplo, un pre-commit para correr ESLint y Prettier (asegurando formato), y un pre-push para ejecutar pruebas unitarias. De este modo, mantenemos la calidad de código constantemente.

- **ci/**: configuraciones para CI/CD, por ejemplo un workflow de GitHub Actions o un pipeline de GitLab que: instale deps, ejecute linter, ejecute pruebas, y de ser posible haga *deploy* continuo a un entorno de staging. Aquí también definiremos variables de entorno necesarias en la plataforma de CI (como URL de API para tests, etc.).

La estructura propuesta garantiza **separación de preocupaciones**: las páginas manejan UI y delegan lógica a servicios/stores; los stores manejan estado; los servicios la comunicación con backend; los componentes de UI la presentación visual. Esto hace que agregar un nuevo módulo (p. ej. campañas) siga el mismo patrón: crear rutas, componentes UI, stores si es necesario, y servicios para llamadas API, sin afectar otras partes.

Además, desde el inicio estamos incluyendo carpetas para **pruebas unitarias y E2E**, enfatizando la importancia del testing. La presencia de `tests/` en la raíz con subcarpetas dedicadas permite escalabilidad de nuestro suite de pruebas a medida que la app crece.

3. Instrucciones Detalladas para el Setup del Entorno

En esta sección se describen los pasos para configurar el entorno de desarrollo, el repositorio, herramientas de calidad de código, CI/CD, variables de entorno y demás aspectos iniciales del proyecto. El objetivo es sentar una base sólida (*Phase 0*) antes de iniciar el desarrollo funcional.

3.1 Inicialización del Repositorio y Proyecto SvelteKit
1. **Crear repositorio**: Iniciar un nuevo repositorio git (por ejemplo, `utalk-frontend`) en la plataforma elegida (GitHub, GitLab, Bitbucket). Añadir un README con descripción del proyecto. 2. **Crear proyecto SvelteKit**: Ejecutar en local:

```
npm create svelte@latest utalk-frontend
cd utalk-frontend
npm install
```

Esto generará la estructura base de SvelteKit. Durante la configuración interactiva, seleccionar TypeScript y ESLint. No es necesario seleccionar un framework de CSS porque añadiremos Tailwind manualmente. 3. **Estructura base**: Verificar que se hayan creado archivos iniciales como `src/routes/+page.svelte` (ejemplo página de inicio), `svelte.config.js`, etc. Podemos eliminar archivos de ejemplo que no usaremos, por ejemplo, la página de ejemplo en `+page.svelte` y su CSS.

3.2 Integración de Tailwind CSS
4. **Instalar Tailwind y dependencias**:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init tailwind.config.js -p
```

Esto crea `tailwind.config.js` y un `postcss.config.cjs`. En el config de Tailwind, configurar la plantilla para que escanee nuestros archivos Svelte:

```
content: ["./src/**/*.{html,js,svelte,ts}"],
```

Agregar presets de shadcn-svelte si se requieren (ver documentación de shadcn-svelte). También configurar colores y fuentes de diseño de UTalk (si ya están definidos). En `tailwind.config.js` se pueden definir las paletas de color corporativo con `theme.extend.colors`. 5. **Configurar SvelteKit + Tailwind**: En el archivo `src/app.css` incluir las directivas base de Tailwind:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Asegurarse de importar `app.css` en el entrypoint de SvelteKit. En SvelteKit v1+, `src/app.html` tiene la línea `%svelte.head%` y `<body>%svelte.body%</body>`. Podemos incluir un `<link rel="stylesheet" href="/app.css">` o importarlo en `src/routes/+layout.svelte`. Una forma común: en `src/routes/+layout.svelte`:

```
<script>
  import "../app.css";
</script>
<slot />
```

Esto aplica Tailwind globalmente. Verificar ejecutando `npm run dev` que Tailwind esté aplicando estilos (probar clases utilitarias en un componente dummy). 6. **shadcn-svelte (componentes UI)**: Seguir la documentación oficial para instalación. Según shadcn-svelte, se puede usar el CLI `npx shadcn-svelte` para incorporar componentes. Por ejemplo:

```
npx shadcn-svelte init
npx shadcn-svelte add accordion button dialog
```

(Esto copiará los componentes seleccionados dentro de nuestra carpeta `src/lib/components/ui/` con sus archivos Svelte, CSS y JS). También es posible instalar su paquete de utilidades UI base (bits-ui). Asegurarse de instalar dependencias necesarias, como `npm i -D @floating-ui/dom` si algunos componentes lo requieren, etc., según las instrucciones.
Tras agregar, revisar que los componentes funcionan y personalizarlos si es necesario (colores de Tailwind ya se ajustarán vía clases, pero podemos editar textos, accesibilidad, etc.). *Nota:* Mantener la versión del CLI actualizada para obtener componentes recientes conforme se necesiten más adelante.

3.3 Configuración de Linter, Formato y Hooks

7. **ESLint**: Si al crear el proyecto seleccionamos ESLint, ya tendremos un `.eslintrc.cjs`. De lo contrario, instalar ESLint y configuraciones:

```
npm install -D eslint eslint-plugin-svelte @typescript-eslint/parser
@typescript-eslint/eslint-plugin
```

Configurar `.eslintrc.cjs` con extends recomendados:

```
extends: [
  "eslint:recommended",
  "plugin:svelte/recommended",
```

```
  "plugin:@typescript-eslint/recommended",
  "prettier"
],
```

Incluir `"overrides"` para archivos Svelte:

```
overrides: [
  {
    files: ["*.svelte"],
    parser: "svelte-eslint-parser",
    parserOptions: { parser: "@typescript-eslint/parser" }
  }
]
```

Añadir reglas personalizadas si deseado (ej. no console.logs en prod, etc.). Activar regla de accesibilidad de Svelte (`"svelte/no-at-html-tags": "error"` , etc., para evitar inyecciones peligrosas). 8. **Prettier**: Instalar y configurar Prettier para formateo consistente:

```
npm install -D prettier prettier-plugin-svelte
```

Crear `.prettierrc` con reglas deseadas (tabWidth, semicolons, etc.). Asegurar en package.json scripts para formateo (`"format": "prettier --write ."`). 9. **Husky & lint-staged**: Instalar Husky para hooks de git:

```
npm install -D husky lint-staged
npx husky install
```

Añadir en package.json:

```
"lint-staged": {
  "*.{ts,svelte,js}": ["eslint --fix", "prettier --write"]
}
```

Configurar husky hooks:

```
npx husky add .husky/pre-commit "npx lint-staged"
npx husky add .husky/pre-push "npm run test"
```

Esto hará que antes de cada commit, se corran linter y prettier autoarreglando el código, y antes de cada push se ejecuten las pruebas. (Ajustar el comando de test según tengamos configurado). 10. **Git Hooks adicionales**: Se puede agregar un hook `commit-msg` para validar mensajes de commit (por ejemplo, usando Conventional Commits). Aunque no es crítico, es buena práctica para CI y claridad en histórico. Si se desea, instalar `commitlint` :

```
npm install -D @commitlint/config-conventional @commitlint/cli
echo "module.exports = {extends: ['@commitlint/config-conventional']}" >
```

```
commitlint.config.js
npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1"'
```

3.4 Variables de Entorno y Configuración
11. **Definir .env**: Crear archivos `.env.development`, `.env.production` con las variables necesarias. Por ejemplo:
- `VITE_API_URL`: URL base de la API backend (diferente en dev/staging/prod si aplica). - `VITE_WS_URL`: URL del servidor WebSocket/Socket.io si es diferente. - Cualquier otra clave pública necesaria (ej: clave pública de servicio de notificaciones, etc.).
Nota: Variables sensibles (ej. clave API de Sentry) se pueden definir sin el prefijo `VITE_` y cargarlas en el server via `$env/static/private`. Variables con `VITE_` quedan expuestas al frontend y se acceden vía `import.meta.env.VITE_API_URL`. 12. **Gestión de variables en SvelteKit**: SvelteKit maneja variables de entorno en build y runtime. Usaremos `$env/static/public` para variables con prefijo `VITE_`. Por ejemplo, en nuestros servicios:

```
import { PUBLIC_API_URL } from '$env/static/public';
```

(si configuramos `VITE_API_URL`, SvelteKit la expone como PUBLIC_API_URL). Para variables privadas (ej. claves secretas para usar en server), usar `$env/static/private`.
Configurar en el README o documentación interna qué variables existen y su propósito, para nuevos desarrolladores.

3.5 Configuración de Testing
13. **Unit Testing (Vitest)**: Instalar Vitest y ambiente de pruebas para Svelte:

```
npm install -D vitest @testing-library/svelte @testing-library/jest-dom jsdom
```

Configurar en `vite.config.ts`:

```
import { sveltekit } from '@sveltejs/kit/vite';
import { defineConfig } from 'vite';
export default defineConfig({
  plugins: [sveltekit()],
  test: {
    environment: 'jsdom',
    globals: true,
    include: ['src/**/*.{test,spec}.{js,ts}']
  }
});
```

Esto permite usar `npm run test` para ejecutar Vitest. Crear un primer test simple, por ejemplo, para el store de usuario: `src/lib/stores/auth.store.spec.ts` para verificar que por defecto el usuario está no autenticado, etc. 14. **E2E Testing (Playwright)**: SvelteKit opcionalmente ofrece plantilla con Playwright. Si no se incluyó al crear el proyecto, podemos añadirlo:

```
npm install -D @playwright/test
npx playwright install  # instala browsers
```

Crear un directorio `tests/e2e`. Escribir un test básico, e.g., `tests/e2e/login.spec.ts` que levante la app (usando `npm run dev` o desplegando a local) y simule: abrir página login, intentar login con credenciales de prueba, verificar redirección al chat. Playwright se integrará en CI para correr en push o mediante GitHub Actions.

1. **Testing Library (Component tests)**: Con @testing-library/svelte instalada, podemos escribir tests de componentes. Por ejemplo, `src/lib/components/ui/Button.spec.ts` para asegurar que nuestro botón renderea el contenido y responde a eventos. Esto refuerza la confianza en la UI.

2. **Coverage**: Configurar Vitest para cobertura de código:

   ```
   npm run test -- --coverage
   ```

   y asegurarnos en CI de publicar el reporte o al menos chequear que el porcentaje mínimo se cumpla (opcional pero deseable para un proyecto crítico).

3.6 Integración Continua (CI) y Despliegue (CD)
17. **CI pipeline**: Configurar la plataforma de CI elegida. Si usamos GitHub, crear `.github/workflows/ci.yml` con un workflow que en cada push o PR: - Instale dependencias (`npm ci`). - Ejecute linter (`npm run lint`) y tests (`npm run test`). - Opcional: ejecute build (`npm run build`) para asegurarse de que la app compila sin errores. - Opcional: si los tests pasan en main, desplegar automáticamente a un entorno (ver siguiente punto). - Artefactos: podemos guardar resultados de cobertura o generar un Storybook (si tuviéramos) para adjuntar, etc. 18. **CD / Deploy**: Decidir dónde se desplegará la aplicación. SvelteKit puede generar un adaptador Node (sirve SSR con un servidor), o podemos desplegar en Vercel/Netlify fácilmente. - Para Vercel: instalar `@sveltejs/adapter-vercel`, configurar en `svelte.config.js` y simplemente push a branch principal para que Vercel haga deploy (o conectar repo a Vercel). - Para Node propio: usar `adapter-node` y desplegar la aplicación como contenedor Docker o proceso PM2. Si es Docker, escribir un `Dockerfile` multi-stage que construya la app (`npm run build`) y luego sirva con `node build`. - **Variables de entorno en prod**: configurar en la plataforma (por ej, en Vercel agregar VITE_API_URL de producción, etc.). Nunca commitear credenciales sensibles.

1. **Habilitar Previews**: En CI, para PRs grandes, configurar despliegues temporales (Vercel preview deploys o similares) para que QA pueda probar funcionalidades antes de mergear a main.

2. **Monitoreo de Deploy**: Preparar integración con Sentry u otra herramienta de monitoreo de errores front-end, pero se activará en fases posteriores antes de producción (ver sección de monitoreo más adelante).

Llegados a este punto, el entorno de desarrollo estará listo: cualquier colaborador puede clonar el repo, ejecutar `npm install` y `npm run dev` y obtener la app base funcionando con Tailwind. Los guardas de calidad (lint/test hooks) funcionan, y el pipeline de CI impedirá merges que rompan el build o los tests. Esto completa la **Fase 0 (Setup)**.

4. Implementación Detallada del Sistema de Login

El sistema de **autenticación (Login)** será la puerta de entrada a UTalk y debe implementarse con especial atención a la seguridad y a la experiencia de usuario. En esta sección se cubren los flujos, formularios, validaciones, manejo de tokens, protección de rutas, reconexión de sesiones y manejo de errores relacionados con autenticación.

4.1 Flujo de Autenticación y UX

- **Pantalla de Login**: Crearemos una página de login simple y clara, accesible vía `/login` (ruta pública). Contendrá un formulario solicitando credenciales, típicamente **email/usuario** y **contraseña**. Si el backend soporta OAuth o login social, se podría añadir posteriormente, pero inicialmente nos centramos en login tradicional. - **Validación de formulario**: Implementar validaciones en el cliente para mejorar UX: - Email con formato válido (regex o input type=email que ya valida parcialmente). - Contraseña no vacía y posiblemente con criterio mínimo (ej. >= 8 caracteres).
Utilizaremos los componentes de formulario de nuestra librería UI (por ejemplo `<Input>` de shadcn-svelte, que ya incluye focus styles accesibles, etc.) y mostraremos mensajes de error bajo los campos si la validación falla. - Botón de *Submit* deshabilitado hasta que el formulario sea válido, para guiar al usuario. - **Proceso de Login**: Al enviar el formulario, SvelteKit ofrece dos enfoques: 1. **Usando Actions (Form Actions)**: En `+page.server.ts` de la ruta login, definiremos un `actions.login` que maneja el POST del formulario. Ventaja: SvelteKit maneja automáticamente la serialización de data y podemos usar `enhance` para progresivos. Este action hará la petición al backend (por ejemplo, `await authService.login(email, password)`). 2. **Usando fetch desde cliente**: Alternativamente, manejar el submit manualmente en el componente (on:submit -> llamar authService y luego navegar). Sin embargo, preferimos actions de SvelteKit por simplicidad y SSR friendly. - **Llamada al Backend**: `authService.login` hará un `POST /auth/login` (o la ruta que indique la documentación del backend) con las credenciales. Debemos manejar: - **Éxito**: El backend probablemente devuelve un token de acceso (JWT) y posiblemente un refresh token, o una cookie. Supondremos JWT para este plan. En caso de éxito: - Guardar los datos de sesión. Usaremos una **cookie HTTPOnly** para almacenar el token de forma segura. Lo ideal: que el backend ya establezca una cookie HttpOnly en la respuesta (por ej., `Set-Cookie: token=jwt; HttpOnly; Secure`). Si el backend no lo hace, podemos nosotros almacenarlo: - Sea estableciendo la cookie en la respuesta usando `cookies.set()` del lado del servidor (dentro del action login). - O, si decidimos no usar cookie, guardar el token en `localStorage` es otra opción, aunque menos segura frente a XSS. **Recomendación**: HttpOnly cookie para JWT (o al menos para refresh token) así JavaScript no puede robarlo [6]. - Guardar también datos del usuario (nombre, email, roles) en `userStore` para uso inmediato en la UI. - Redirigir al usuario a la página principal de la app (por ejemplo, `/chat` o dashboard). - **Error**: Si las credenciales son inválidas, mostrar un mensaje de error claro ("Usuario o contraseña incorrectos"). Si ocurre un error de red o servidor, mostrar mensaje genérico ("Error de servidor, intente más tarde"). - Estas notificaciones pueden mostrarse mediante un componente de alerta (por ejemplo, un `<Alert>` de shadcn-svelte) o simplemente un texto estilizado rojo bajo el formulario. - Asegurarse de no revelar información sensible en los mensajes (no decir "usuario existe o no", solo "credenciales no válidas"). - **Indicador de carga**: Durante el proceso de login (mientras espera respuesta), mostrar feedback: deshabilitar botón, quizás mostrar un spinner "Conectando...". Esto mejora la percepción de velocidad.

4.2 Gestión de Tokens y Sesión

- **JWT y Refresh**: Suponiendo uso de JWT de duración limitada, implementaremos la lógica de *refresh token*: - El backend podría proporcionar dos tokens: **accessToken** (corto, ej. 15min) y **refreshToken** (largo, ej. 7 días). El refresh token idealmente se guarda en HttpOnly cookie también. - El cliente deberá, al acercarse la expiración del accessToken o recibir un 401, solicitar un nuevo token (`POST /auth/refresh` con el refreshToken). - Implementar en `auth.service.ts` una función `refreshToken()`

que: - Verifique si tenemos refresh token disponible (cookie o almacenamiento). - Haga la petición al backend para renovar el accessToken. - Actualice la cookie/token en memoria y el `userStore` en caso de éxito. - Si falla (refresh expirado), forzar logout global. - **Almacenar sesión**: Vías posibles: - Cookies HttpOnly (preferido): El login action establece `cookies.set('session', jwt, { httpOnly: true, secure: true, path: '/' })`. SvelteKit permitirá leer esa cookie en cada request SSR. - `userStore` en Svelte: Mantendrá el estado de login en la aplicación cliente (por ejemplo `userStore.isAuthenticated = true`). Este store se repuebla en SSR leyendo la cookie (ver siguiente punto). - **Persistencia cross-session**: Si el usuario cierra la pestaña o refresca la página, queremos que siga logueado si su token es válido. Con cookie HttpOnly, esto es automático en SSR: cada request incluirá la cookie, y el hook del servidor podrá validar y restaurar la sesión. - Si no usamos SSR (app como SPA pura), y guardamos token en localStorage, al cargar la app deberíamos leer localStorage y validar token con backend o decodificar JWT para restaurar sesión. - **Implementación recomendada**: Utilizar el **hook** `src/hooks.server.ts` de SvelteKit para autenticar en cada request. Por ejemplo, en `hooks.server.ts`:

```
import { decodeJWT, getUserData } from '$lib/server/auth'; // funciones
backend or library
export const handle: Handle = async ({ event, resolve }) => {
  const token = event.cookies.get('session');
  if (token) {
    try {
      const userData =
decodeJWT(token); // valida firma JWT (o llama backend to verify)
      event.locals.user = userData; // adjuntamos info de usuario a la
request
    } catch(e) {
      // Token inválido o expirado
      event.locals.user = null;
    }
  }
  const response = await resolve(event);
  return response;
};
```

De esta forma, cualquier ruta puede saber si `event.locals.user` existe para determinar autenticación. (Si usamos sesiones de servidor, podría haber otro mecanismo, pero JWT así lo permite sin estado servidor). - **Protección de rutas (guards)**: - Con la estrategia anterior, podemos crear un **guard global** en el hook: si `event.locals.user` es `null` y la ruta solicitada requiere auth, redirigir a `/login`. Por ejemplo:

```
if (!event.locals.user && event.url.pathname.startsWith('/chat')) {
  throw redirect(303, '/login');
}
```

Se puede mantener una lista de rutas públicas permitidas y redirigir todo lo demás (como en el ejemplo implementado) [7]. De esta manera, no se puede acceder a rutas internas sin haber iniciado sesión. - A nivel de interfaz cliente, también ocultaremos o evitaremos navegación a secciones privadas si no hay sesión. Por ejemplo, no mostrar enlaces de menú de CRM/chat si no está autenticado. Pero la seguridad real la hace el guard del servidor. - En el caso de SvelteKit funcionando en modo SPA (sin SSR),

podríamos tener un guard en el cliente (ej. un `$: if (!loggedIn) goto('/login')` en layout), pero dado que usamos SSR/hook, cubrimos ambos. - **Redirección post-login**: Si estando autenticado se intenta ir a `/login`, conviene redirigir a home/chat para evitar volver a login innecesariamente (esto se muestra en el código de ejemplo que redirige si user ya está logueado) [8]. - **Logout**: - Implementar un mecanismo de logout claro. Por ejemplo, un botón "Cerrar sesión" en el menú de usuario. Al hacer clic: - Remover credenciales: borrar la cookie con `cookies.delete('session')` (vía un action en SvelteKit) o si es localStorage, remover el item. - Opcional: notificar al backend para invalidar token (algunos backends mantienen lista de tokens revocados). - Limpiar `userStore` a null. - Redirigir a `/login` y quizás mostrar notificación "Has cerrado sesión". - Asegurarse de manejar logout también si el refresh token falla (sesión expirada).

4.3 Manejo de Errores y Estados Especiales
- **Errores de Autenticación**: Ya mencionado, mostrar mensajes en UI para: - Credenciales incorrectas. - Cuenta bloqueada (si backend devuelve un código específico). - Usuario no verificado (si hubiera verificación por email). - Estos casos se manejan inspeccionando el error devuelto por `authService.login` (códigos HTTP 401, 403, etc.) y asignando mensajes de error en el estado local del formulario. - Usar un componente de alerta accesible (aria-live polite para que lectores de pantalla anuncien el error). - **Timeout / Expiración de sesión**: Si el usuario está inactivo o el token expira: - Si implementamos refresh token, la sesión podría renovarse automáticamente. Pero supongamos que incluso el refresh expira (por ejemplo tras días sin usar): - La próxima petición obtendrá 401; en el hook o en fetch interceptors del `api.ts` podemos detectar 401 y trigger logout. - Podemos mostrar un mensaje "Tu sesión expiró, por favor inicia sesión de nuevo" en la pantalla de login tras redirigir. - Para mejorar UX, podríamos implementar un contador de inactividad: una función que después de X horas muestre un popup "¿Sigues ahí?" antes de cerrar sesión. Esto es opcional. - **Reinicio/Recuperación de contraseña**: No solicitado explícitamente, pero vale mencionarlo: - Dejar preparado un enlace "¿Olvidaste tu contraseña?" en la pantalla de login, que eventualmente apunte a un módulo de recuperación (flujo de enviar email, etc.). No se implementará ahora, pero se considera en diseño (un pequeño detalle mejora la integridad del sistema de login). - **Registro de usuarios**: Tampoco se pidió, pero si UTalk admite registro: - Considerar un flujo de signup similar al login, en ruta `/register` con su formulario, validaciones (por ej. fuerza de password), y llamadas a backend. - La arquitectura actual lo permitiría fácilmente añadiendo `routes/register` y un método `authService.register()`. No nos enfocamos en implementarlo ahora, pero tenerlo en mente asegura que nada del login actual lo impide (por ejemplo, la cookie puede ser la misma "session", etc.).

4.4 Re-conexión de Sesión Automática
- Si la app pierde conexión a internet momentáneamente, podríamos enfrentarnos a llamadas fallidas al backend. Para mejorar la resiliencia: - Implementar en `api.ts` re-intentos automáticos para ciertas peticiones idempotentes (quizá no para login, pero sí para refresh). - En el caso del chat en particular, trataremos la reconexión de socket en la sección de chat. Pero en contexto login: si el usuario se logueó y luego el token expira mientras está offline, al reconectar la app debería intentar el refresh inmediatamente (quizá vía Service Worker o apenas recupere conexión). - El navegador provee eventos `window.navigator.onLine` que podemos escuchar en un store "networkStatus". Si detectamos reconexión y sabemos que el token expiró, podríamos llamar refresh de inmediato. - De todos modos, con cookies HttpOnly, en cuanto vuelva la conexión el próximo request SSR podría refrescar token con ayuda del backend (si implementado allá). Esto ya entra en lógica de backend, pero lo mencionamos. - En suma, la “reconexión” en login se refiere a mantener al usuario autenticado durante cortes breves de red o al reabrir la aplicación mientras la sesión sigue válida, lo cual cubrimos con refresh token y cookies persistentes.

4.5 Seguridad en Autenticación
La seguridad es crítica en login: - **Almacenamiento seguro**: Reiterando, HttpOnly cookies siempre que

sea posible para tokens, mitigando XSS. Si se usa localStorage, entonces extremar sanitización de inputs para no introducir XSS que robe tokens. - **Comunicación**: Siempre usar conexiones HTTPS para llamadas de login y cualquier envío de credenciales. Activar `secure: true` en cookies para que solo viajen por HTTPS. - **Protección contra CSRF**: Si usamos cookies para auth, debemos proteger las acciones sensibles (login, refresh) de CSRF: - SvelteKit actions ya nos permiten usar el método POST (lo cual no es suficiente por sí solo), pero podemos implementar un *double submit cookie* or *SameSite=strict* en la cookie. Si `SameSite=Lax` o `Strict`, la cookie no se envía en requests cross-site, mitigando CSRF en la mayoría de casos. - Otra técnica: usar un token CSRF (por ejemplo, al servir la página login, incluir un token CSRF en un hidden input, y validarlo en el action). - Dado que UTalk es una app principalmente de primer partido (usuario accede directamente), `SameSite=Lax` podría ser suficiente. En config de cookie, añadir `sameSite: 'lax'`. - **Limitar intentos**: A nivel frontend, podríamos poner un simple bloqueo tras X intentos fallidos (e.g., deshabilitar el form por 30 segundos después de 5 intentos, mostrando mensaje). Sin embargo, eso es fácil de sortear y realmente se debe controlar en backend (rate limit por IP). Lo mencionamos para coordinación con backend más que implementar en front. - **Logging out on multiple tabs**: Si el usuario cierra sesión en una pestaña, idealmente otras pestañas también deberían detectar y cerrar. Con cookies, esto sucede automáticamente en la siguiente acción. Pero para ser más proactivo en UI: - Podemos usar el Storage event: si usamos localStorage for tokens, un logout que hace `localStorage.removeItem('token')` en una pestaña dispara `window.onstorage` en otras. Podemos suscribirnos a eso y cerrar sesión allí también. - Con cookies HttpOnly puras, no hay una notificación, pero cada pestaña al hacer la siguiente request SSR verá que no hay token y redirigirá a login. Para mejora, podríamos usar BroadcastChannel API entre tabs para notificar logout, pero es refinamiento futuro.

Al completar la implementación de login con los puntos anteriores, tendremos un sistema robusto donde el usuario puede iniciar sesión de forma segura, la sesión se mantiene mientras corresponda, y las rutas internas de la aplicación están protegidas de accesos no autorizados. Esto cubre la **Fase 1 (Login completo)**.

5. Implementación del Módulo de Chat en Tiempo Real

El módulo de **Chat** es el corazón de UTalk, permitiendo envío y recepción de mensajes (incluyendo multimedia), notificaciones en tiempo real, e interacciones fluidas entre usuarios. Este apartado detalla cómo construir el chat con SvelteKit y Socket.io (u WebSockets), manejando estados de conexión, reconexión automática, indicadores de envío, y asegurando rendimiento y seguridad.

5.1 Arquitectura de Comunicación en el Chat

- **Socket de tiempo real**: UTalk utilizará sockets para actualizar mensajes en vivo. Asumiremos que el backend expone un endpoint de WebSocket (por ejemplo con Socket.io en `wss://api.utalk.com/chat`). Usaremos la librería cliente Socket.io en el frontend por su facilidad: - Al entrar al módulo de chat, establecemos la conexión:

```
import { io, Socket } from 'socket.io-client';
let socket: Socket;
socket = io(PUBLIC_WS_URL, {
  auth: { token: userJWT }  // enviar token JWT para autenticación en
handshake
});
```

(Asegurarse de usar el token actual del usuario para que el backend autentique la conexión). - Manejaremos eventos: `socket.on('connect', ...)`, `socket.on('message', ...)`, etc., según protocolo definido en backend docs. Por ejemplo, el backend podría emitir evento `"message"` cada vez que llega un nuevo mensaje para el usuario. - Guardaremos la instancia `socket` en nuestro `chat.store.ts` o en un módulo singleton para reutilizarlo mientras el usuario esté en chat. - **Estados en UI**: Distinguiremos principales estados: - **Conectando**: Socket intentando conectar (mostrar en UI algo como "Conectando..." quizás un icono de estado gris). - **Conectado**: Socket abierto (mostrar estatus verde o "En línea"). - **Desconectado/Reintentando**: Si se pierde la conexión, Socket.io automáticamente intentará reconectar. Informar al usuario con un aviso ("Reconectando..."). Podemos desactivar entrada de texto mientras no hay conexión para evitar mensajes perdidos, o almacenarlos en cola. - **Recepción de Mensajes**: Cada mensaje recibido por socket será inmediatamente reflejado en la UI (añadido a la lista de mensajes). - **Envío de Mensajes**: Al enviar un mensaje, optimísticamente podríamos mostrarlo en la lista con un estado "enviando" (por ejemplo, opacidad menor o un spinner pequeño al lado) hasta que el servidor confirme. Si la confirmación llega (podría ser un ACK por socket), actualizamos su estado a enviado/entregado. - **Estados de entrega/lectura**: Si el backend envía confirmaciones de lectura (ej. "usuario X leyó hasta mensaje Y"), debemos actualizar los mensajes con un indicador (ej. doble check estilo WhatsApp). Esto implica que el mensaje modelo tiene un campo `status` (enviado, entregado, leído). Manejaremos estos eventos también en el socket (por ej. `socket.on('message_status', ...)`). - **Usuarios escribiendo (typing)**: Si requerido, el socket puede emitir eventos "typing" cuando alguien está escribiendo. Podemos mostrar "Fulano está escribiendo..." en la UI. Implementarlo haciendo que al detectar input en el chat, enviemos un pequeño evento (limitado por tiempo para no saturar). - **Almacenamiento de Mensajes**: - En el `chat.store` tendremos una `messages` store (array de mensajes actuales en la conversación activa). Este store se actualiza cuando: - Entramos a una conversación (cargamos historial desde API). - Llega un nuevo mensaje por socket (hacemos `messages.update()` para agregarlo). - Enviamos un mensaje (podemos agregarlo de una vez con estado "pending"). - Recibimos confirmación/ack (actualizamos el objeto mensaje en el array, cambiando su estado). - Para persistencia ligera, podríamos utilizar **IndexedDB** o `localStorage` para cachear últimos mensajes, de manera que si se recarga la página los mensajes recientes aún aparezcan mientras se obtienen del servidor. Sin embargo, esto es un extra; inicialmente podemos confiar en recargar historial vía API. - Paginación: No cargaremos miles de mensajes de golpe. Implementaremos carga incremental: - Mostrar por ejemplo los últimos 50 mensajes. Si el usuario hace scroll hacia arriba (al inicio de la lista), cargar 50 más antiguos (vía llamada REST `GET /conversations/{id}/messages?before=msgId`). - Svelte: se puede detectar cuándo el scroll llega al top y entonces disparar la carga de más mensajes, agregándolos al inicio de la lista. Al hacer esto, mantener la posición de scroll para no saltar abruptamente. - Este enfoque asegura que la aplicación pueda manejar historiales grandes sin problemas de rendimiento (solo carga lo necesario bajo demanda).

5.2 Envío de Mensajes y Multimedia

- **Componente de Input de Chat**: Tendremos un componente `ChatInput.svelte` con: - Un `<textarea>` o campo de texto adaptable para el mensaje. - Botón de enviar. - Botón de adjuntar archivo (clip). - Posiblemente iconos para emojis (según alcance, pero se puede dejar para después). - Este componente manejará el estado del mensaje que se está escribiendo. - Al enviar (submit), emitirá un evento o llamará directamente `chatService.sendMessage(conversationId, content)`. - **Lógica de envío**: - `chatService.sendMessage(convId, content)`: - Verificar que `socket.connected` es true (si no, se puede: o bien reintentar más tarde, o guardar en una cola temporal hasta reconexión). - Emitir por socket: `socket.emit('message', { convId, content, type: 'text' })`. (El protocolo puede variar; podríamos necesitar incluir un ID temporal para seguimiento). - Inmediatamente reflejar en UI: crear un objeto `Message` con `id: 'temp-id'`, `content`, `sender: yo`, `status: 'sending'`, `timestamp: now`, y agregarlo al store

`messages`. Esto da la ilusión de inmediatez. - Cuando llegue la confirmación del servidor, que probablemente incluirá el ID real del mensaje y quizá cambie status a 'sent': - Actualizar el mensaje con ID temporal en el store reemplazándolo por el definitivo o marcando status enviado. Para identificarlo, podemos usar un campo `tempId` correlacionado. - Si tras un tiempo no hay confirmación (por ej., 5 segundos), marcar ese mensaje como error (status 'failed') y ofrecer reintentar (podemos hacer que al tocar un mensaje en error reenvíe). - **Edge case**: Si el socket está desconectado en el momento de enviar, podemos optar por deshabilitar el botón de enviar (mejor UX: el usuario ve que no puede enviar hasta reconectar), o permitir cola: - Si permitimos cola: almacenar mensajes en un array `pendingMessages`. Cuando el socket reconecte, enviar todos los pendientes. Esta es una mejora que podemos planear. - Indicar claramente cuando no hay conexión ("No conectado, intentando reconectar...") para que el usuario entienda por qué no se envían sus mensajes. - **Adjuntar Archivos**: - Al hacer clic en adjuntar, se abre un `<input type="file" accept="image/*, video/*, etc.">`. Soportaremos imágenes inicialmente, pero la arquitectura permitirá más tipos (audio, PDF, etc. según necesidades). - Al seleccionar un archivo, usaremos `file.service.ts` para la carga: - Opción A: **Subida directa via REST** antes de enviar mensaje. Ej: `const fileUrl = await fileService.upload(file)`. Este servicio podría: - Llamar a `POST /files` con el blob, incluyendo el token de auth (lo enviará la cookie o Authorization header). - Recibir de respuesta un `fileId` o URL pública. - Retornar ese identificador. - Luego emitimos el mensaje vía socket con `type: 'image'` y la referencia al archivo (`fileId` o URL). El servidor al recibirlo ya sabrá cómo distribuirlo. - Opción B: **Enviar archivo por el socket** directamente. Socket.io permite enviar binarios. Esto ahorra la llamada REST, pero puede ser menos eficiente para archivos grandes. Probablemente el backend ya define cómo hacerlo; muchos sistemas prefieren REST for files para manejar almacenamiento. - Adoptaremos opción A por claridad: subida REST, luego mensaje socket con metadata. - **UI de subida**: Mientras el archivo se sube, mostrar un progreso o al menos un estado "subiendo archivo...". Tailwind puede estilizar barras de progreso, o un spinner sobre la vista previa. - Podríamos generar una URL local de previsualización (usando `URL.createObjectURL(file)`) para mostrar la imagen seleccionada incluso antes de subir, mejorando la UX. - Si la subida falla (problema de red), notificar al usuario y permitir reintentar. - **Recepción de archivos**: Cuando llega un mensaje tipo imagen, la UI mostrará la imagen. Asegurarse de: - Usar `<img>` con `src={fileUrl}` y alt adecuado (por accesibilidad, alt podría ser "Imagen enviada por X" o extraer nombre del archivo si lo hay). - Para otros tipos (pdf, etc.), mostrar un icono y un enlace para descargar. - Videos/audio podrían tener reproductores HTML5 integrados. - **Optimización de multimedia**: - Implementar *lazy loading* de imágenes (atributo `loading="lazy"` en <img>), así imágenes fuera de viewport no cargan hasta hacer scroll. - Podríamos limitar el tamaño de video a reproducir automáticamente, etc., pero eso se manejará caso a caso. - Si los archivos se sirven desde un CDN o servidor con tamaño grande, confiar en su throughput, pero podemos más adelante añadir indicadores de descarga (por ejemplo, blur de imagen hasta que cargue completamente). - **Multimedia thumbnails**: Si hay imágenes, podríamos implementar un visor (lightbox) al hacer clic para ver en tamaño completo. Dejamos esa mejora como futura.

5.3 Manejando Estados de Carga y Feedback al Usuario

En un chat en tiempo real, el feedback visual es clave: - **Estado "typing..."**: Como mencionado, mostrar cuando el otro usuario escribe. Implementación: - Cada vez que el usuario actual escribe, después de X ms de pausa, enviar `socket.emit('typing', { convId, isTyping: true })`. Al dejar de escribir o enviar, emitir `isTyping: false`. - Recibir `socket.on('typing', data => { who, convId } )` y si `who` no es el current user, actualizar un pequeño estado en UI (ej. a nivel ChatFooter: "User X está escribiendo..."). - Debe haber un *debounce* para no flickear. Por ejemplo, si no recibimos 'typing' en 5s, asumimos que dejó de escribir. - **Estado de entrega**: Indicar a qué mensajes se les dio entrega y lectura: - Entrega (delivered) podría mostrarse con un check simple, lectura (read) con doble check pintado, similar a WhatsApp. Esto depende de que backend envíe esos eventos (p.ej., evento `message_delivered` o `message_read` con ids). - Actualizar el `messages` store marcando los que correspondan. Si conocemos el usuario receptor abrió la conversación, posiblemente marcar todos

como leídos. - **Tiempos y Fecha**: Mostrar timestamp relativo en cada mensaje (ej. "10:45 am"). Y separadores de fecha (ej. "Hoy", "Ayer" cuando corresponda). Esto se hace en la UI (component MessageList agrupa por día, puede insertar un separador). - Utilizar utils de fecha para formatear legiblemente. - **Scroll y auto-scroll**: - Cuando llega un mensaje nuevo y el usuario ya está al final del chat, hacer auto-scroll hacia abajo para mostrarlo. - Si el usuario está viendo mensajes antiguos (scroll up) y llega uno nuevo, quizás mostrar un botón "Ir al último mensaje" en lugar de forzar scroll (para no sacar de contexto). - Podemos detectar posición con `element.scrollHeight - element.scrollTop === element.clientHeight` (scrolled to bottom). - **Indicadores de carga de historial**: - Cuando cargamos mensajes antiguos (scroll up), mostrar un spinner en el top mientras se cargan. - Si la conversación es muy larga, después de cierto scroll podemos ofrecer un botón "Cargar más..." manual en lugar de infinito, para no saturar.

5.4 Reconexión Automática del Socket

La conexión de chat debe ser resiliente a desconexiones: - Socket.io por defecto intenta reconectar con backoff exponencial. Aun así, haremos: - Escuchar evento `socket.on('disconnect', reason)`: cuando se desconecta, actualizar estado `chatStore.connection = 'disconnected'`. También posiblemente detener ciertas UI (ej. deshabilitar input). - En el mismo handler, si `reason !== 'logout'` (o no fue intencional), mostrar "Reconectando..." y dejar que Socket.io intente. Podemos incluso llamar `socket.connect()` manual tras algunos segundos si quisiéramos más control. - Escuchar `socket.on('reconnect_attempt'...)` o `reconnecting` para informar estado. - Escuchar `socket.on('reconnect_failed')` para eventualmente dar opción de reconectar manual (por ejemplo, mostrar un botón "Reconectar"). - Cuando reconecta (`socket.on('connect'` tras desconexión), actualizar estado a 'online' y si hay mensajes pendientes enviar. También quizás refetch del historial para captar mensajes perdidos durante la caída: - Socket.io tiene un parámetro `withCredentials` y mecanismos de buffer. Pero por seguridad, lo más robusto: tras reconectar, pedir al backend los mensajes recientes desde el último que teníamos, para no perder ninguno. E.g., keep track del last message ID, y hacer `GET /conversations/{id}/messages?after=lastId` y fusionar con store. Así cubrimos incluso casos de mensajes emitidos mientras el usuario estaba offline. - **Duplicación de mensajes**: a veces durante reconexiones, se puede recibir duplicados (ej. si no filtramos ya recibidos). Mantener un set de IDs de mensajes en el store para ignorar duplicados al fusionar. - **Cambio de estado de conexión en UI**: Por transparencia, podemos mostrar un pequeño indicador (un punto verde para conectado, rojo desconectado). Ubicación: quizá en la cabecera del chat o al lado del nombre de usuario. Herramientas de UI (shadcn) pueden ofrecer un Badge/Status component para esto.

5.5 Optimización de Performance en Chat

- **Virtualización de lista de mensajes**: Si un chat tiene miles de mensajes, montar todos en el DOM puede ser pesado. Considerar uso de un componente de *virtual list* (existen algunos para Svelte, o podemos implementar básico). Virtualization solo renderiza lo visible. Esto podría implementarse en fases posteriores si vemos problemas de rendimiento con ~100+ messages. Inicialmente, la paginación de historial ya limita la carga. - **Evitar re-render masivo**: Svelte re-renderiza reactivo. Si actualizamos un mensaje en medio de la lista (por status), por defecto reenviará la lista entera. Mitigar: - Usar indices o keys apropiadas en `{#each messages as msg (msg.id)}` para que Svelte reconozca identidad de elementos. - Dividir MessageList en subcomponentes (MessageItem) para localizar las actualizaciones. - Considerar si needed usar `immutable` data patterns (pero Svelte is fine with local changes in array if we do store.update). - **Throttle UI updates**: Si llegaran decenas de mensajes por segundo (caso raro), podríamos throttle actualizaciones agrupándolas. Probablemente innecesario en un chat normal. - **WebSocket traffic**: Ensure not to do heavy computations on each message event. Our message handling will be straightforward append; anything heavy (like processing an image) should be done in a WebWorker if needed (not likely needed). - **Memory management**: Si el usuario abre muchas conversaciones, quizá descargar de memoria las no activas. Por ejemplo, podríamos store solo current

conversation messages, and if user switches conv, dump or keep only a summary of others. Alternatively, if multi-chat UI (like Slack with multiple DMs), we'd need more caching. For now, assume one chat view at a time (like WhatsApp style: select conversation, see messages). - **Lazy load modules**: Ensure that modules like CRM, etc., that are not used in chat, are code-split. SvelteKit does this automatically by route. But for chat itself, if it becomes large, we could split some parts (like maybe heavy image viewer component could be dynamic import).

5.6 Seguridad en el Chat (sanitización y restricciones)
- **Sanitización de contenido**: As chat acepta input de usuarios, debemos sanitizar para evitar XSS: - Si permitimos texto enriquecido (links clicables, emoji, etc.), hay que escapar cualquier HTML. Por defecto, insertar variables en Svelte (`{@html ...}`) sería peligroso; en cambio, debemos manejar todo como texto. Svelte escapará las interpolaciones normales, así que si `message.content = "<script>..."`, no ejecutará, solo mostrará como string, lo cual es seguro. - Para mayor control, podríamos usar una librería de sanitización (DOMPurify) si en algún caso renderizamos HTML (por ejemplo, si en futuras se permite formatear mensajes tipo markdown a HTML). - También conviene filtrar en backend, pero en front no está de más. - **Límite de tamaño**: Prevenir crashes o abusos: - Limitar el tamaño de mensaje de texto (ej. 1000 caracteres) con validación al enviar. - Limitar tipos y tamaño de archivos adjuntos (ej. no permitir > 10MB en frontend, mostrando error "Archivo demasiado grande"). - Estas restricciones deben ser espejo de las del backend para una experiencia consistente. - **Contenido multimedia seguro**: - Imágenes: podrían potencialmente contener malware (aunque improbable afectar web). Aun así, cargar imágenes de orígenes confiables (nuestro servidor/CN). Incluir atributo `referrerpolicy="no-referrer"` si mostramos imágenes de enlaces externos, para no filtrar tokens de acceso en la cabecera Referer. - Videos: no auto reproducir audio sin acción del usuario (good UX and to avoid any malicious loud audio). - **Protección de datos sensibles**: Si el chat puede enviar información sensible, quizás cifrado de extremo a extremo sería ideal, pero es un nivel fuera del alcance por ahora (y manejarlo requeriría encryption keys en front). Mencionamos que no implementamos eso de momento, pero mantenemos los canales seguros (SSL). - **Auditoría**: Para completitud futura, contemplar un log de chat en cliente es innecesario (server se encarga), pero podríamos habilitar algún debug mode para registrar eventos en consola para debugging.

Tras implementar el chat con todo lo anterior, tendremos un **Módulo de Chat completo**: el usuario puede enviar y recibir mensajes instantáneamente, adjuntar imágenes, ver estados de envío y lectura, y todo esto incluso si la conexión fluctúa, manteniendo la integridad de la conversación. Esto corresponde a **Fase 2, 3 y 4** en gran medida (dependiendo de cómo definamos los hitos, a continuación se hará).

6. Ejemplos de Código (Estilo *IA-Friendly*)

A continuación, se muestran fragmentos de código ilustrativos del proyecto UTalk, escritos con un estilo claro, **nombres descriptivos**, y comentarios explicativos. El objetivo es que incluso una inteligencia artificial de asistencia (como Cursor AI) o un nuevo desarrollador puedan leer y entender rápidamente la intención y funcionamiento del código. Estos ejemplos reflejan las convenciones que usaremos en todo el código base.

6.1 Store de Autenticación (`auth.store.ts`)

Este store global mantiene el estado de la sesión de usuario (datos básicos y estado de login). Emplea un `writable` de Svelte y expone funciones para actualizar la sesión de forma consistente. Los comentarios explican cada parte del código:

```ts
import { writable } from 'svelte/store';

/** Datos del usuario autenticado */
interface User {
  id: string;
  name: string;
  email: string;
  // otros campos relevantes...
}

/** Estado de autenticación que mantiene el store */
interface AuthState {
  user: User | null;
  token: string | null;
  isAuthenticated: boolean;
}

/** Estado inicial: sin usuario logueado */
const initialState: AuthState = {
  user: null,
  token: null,
  isAuthenticated: false
};

/**
 * authStore: almacena y gestiona el estado de autenticación.
 * Se puede suscribir para obtener el usuario actual y si está autenticado.
 */
function createAuthStore() {
  const { subscribe, update, set } = writable<AuthState>(initialState);

  return {
    subscribe,
    /** Establece los datos de usuario tras un login exitoso */
    setSession: (userData: User, token: string) => {
      set({ user: userData, token, isAuthenticated: true });
    },
    /** Elimina los datos de sesión (logout) */
    clearSession: () => {
      set(initialState);
    },
    /** Actualiza parcialmente el usuario (ej. cambio de perfil) */
    updateUser: (newUserData: Partial<User>) => {
      update(state => {
        if (!state.user) return state;
        return { ...state, user: { ...state.user, ...newUserData } };
      });
    }
  };
}
```

```
// Exportamos una instancia única del store de auth
export const authStore = createAuthStore();
```

En este ejemplo: - Los nombres de variables e interfaces son claros (`AuthState`, `setSession`, `clearSession`). - Cada método tiene un comentario JSDoc describiendo su finalidad. - Se muestra cómo manejar un caso de actualización parcial del usuario. - Este estilo facilita a cualquier lector (humano o IA) entender rápidamente cómo manipular la sesión.

6.2 Servicio de Chat (`chat.service.ts`)

Ejemplo simplificado del servicio de chat, encargándose de conectar con el WebSocket y gestionar envío/recepción de mensajes. Incluye comentarios paso a paso:

```
import { io, Socket } from 'socket.io-client';
import { chatStore } from '$lib/stores/chat.store';
import { PUBLIC_WS_URL } from '$env/static/public';  // URL del servidor de
WebSocket

let socket: Socket | null = null;

/**
 * Inicia la conexión del socket de chat usando el token JWT para
autenticación.
 * Guarda eventos para manejar mensajes entrantes y cambios de estado.
 */
export function connectChatSocket(token: string): void {
  socket = io(PUBLIC_WS_URL, {
    auth: { token },
    autoConnect: false  // preferimos conectar manualmente
  });
  // Intentamos conectar
  socket.connect();

  socket.on('connect', () => {
    console.log('  Socket conectado al servidor de chat');
    chatStore.setConnectionStatus('connected');
    // Opcional: fetch de mensajes recientes si es reconexión
  });

  socket.on('disconnect', (reason) => {
    console.warn('⚠ Socket desconectado:', reason);
    chatStore.setConnectionStatus('disconnected');
    // Socket.io reintentará automáticamente a menos que reason sea "io
client disconnect"
  });

  socket.on('connect_error', (err) => {
    console.error('  Error conectando socket:', err.message);
    // Podemos actualizar estado a "error" si deseamos manejar distinto a
```

```
disconnect normal
    chatStore.setConnectionStatus('error');
  });

  // Manejar evento personalizado 'message' de nuevo mensaje entrante
  socket.on('message', (msg) => {
    // Se asume msg tiene estructura { id, content, senderId,
timestamp, ... }
    chatStore.addMessage(msg);
    // Podríamos agregar lógica de notificación aquí si la conversación
actual no está en foco
  });

  // Manejar confirmación de entrega de mensaje enviado (ACK)
  socket.on('message_ack', (ack) => {
    // ack podría contener el id temporal y el nuevo id asignado
    chatStore.confirmMessageSent(ack.tempId, ack.finalId, ack.timestamp);
  });

  // Manejar evento de usuario escribiendo
  socket.on('typing', ({ conversationId, userId, isTyping }) => {
    chatStore.setUserTyping(conversationId, userId, isTyping);
  });
}

/**
 * Envía un mensaje de texto a través del socket.
 * Retorna el ID temporal asignado al mensaje mientras se confirma.
 */
export function sendMessage(conversationId: string, content: string): string
| null {
  if (!socket || !socket.connected) {
    console.error('No se pudo enviar, socket desconectado');
    return null;
  }
  // Creamos un ID temporal para el mensaje
  const tempId = `temp-${Date.now()}`;
  const messagePayload = { tempId, conversationId, content, type: 'text' };
  socket.emit('message', messagePayload);
  // Registramos el mensaje en el store con estado "sending"
  chatStore.addMessage({
    id: tempId,
    conversationId,
    content,
    senderId: chatStore.currentUserId(),
    timestamp: new Date().toISOString(),
    status: 'sending'
  });
  return tempId;
}
```

```
/** Envia notificación de "estoy escribiendo" al servidor */
export function sendTyping(conversationId: string, isTyping: boolean): void {
  if (!socket) return;
  socket.emit('typing', { conversationId, isTyping });
}

/** Cierra la conexión del socket de chat (por ejemplo, al hacer logout) */
export function disconnectChatSocket(): void {
  if (socket) {
    socket.disconnect();  // causa un disconnect manual (socket.io no
reintentará)
    socket = null;
  }
}
```

Características a notar: - Uso de `console.log` con emojis (, ⚠,) para facilitar depuración visual durante desarrollo. - Funciones pequeñas con una responsabilidad clara (conectar, enviar mensaje, enviar typing, desconectar). - Comentarios explican qué hace cada bloque, por ejemplo al recibir 'message_ack'. - Se utilizan métodos del `chatStore` (que imaginamos implementa `addMessage`, `confirmMessageSent`, etc.) para mantener la lógica de estado centralizada. - Buen manejo de casos: no envía si el socket no está conectado, retorna null en ese caso (el llamante podría manejar ese null mostrando error UI).

6.3 Componente de Login Form (`+page.svelte` en login)

Un fragmento de cómo sería el componente Svelte para el formulario de login, usando un action de SvelteKit. Incluimos comentarios dentro del markup:

```
<script lang="ts">
  import { invalidate } from '$app/navigation'; // para refrescar actions
  import { authStore } from '$lib/stores/auth.store';
  export let form; // resultado del action (éxito o error)

  let email: string = "";
  let password: string = "";

  // Opcional: derivar error desde form si vino con error del servidor
  $: loginError = form?.error ? form.error.message : null;
</script>

<main class="min-h-screen flex items-center justify-center bg-gray-50">
  <form method="POST" action="?/login" class="bg-white p-6 rounded shadow-sm
w-full max-w-md"
        on:submit={() => invalidate('login')}>
    <h1 class="text-2xl font-bold mb-4 text-center">Iniciar Sesión</h1>
    <!-- Campo de Email -->
    <div class="mb-3">
      <label for="email" class="block text-sm font-medium mb-1">Email</label>
      <input id="email" name="email" bind:value={email} type="email"
required
```

```
            class="input bg-gray-50 border border-gray-300 rounded w-full
px-3 py-2" />
    </div>
    <!-- Campo de Contraseña -->
    <div class="mb-4">
      <label for="password" class="block text-sm font-medium
mb-1">Contraseña</label>
      <input id="password" name="password" bind:value={password}
type="password" required
            class="input bg-gray-50 border border-gray-300 rounded w-full
px-3 py-2" />
    </div>
    <!-- Mensaje de error si existe -->
    {#if loginError}
      <div class="text-red-600 text-sm mb-3" role="alert">
        {loginError}
      </div>
    {/if}
    <!-- Botón Submit -->
    <button type="submit" class="btn bg-blue-600 text-white w-full py-2
rounded hover:bg-blue-700 disabled:opacity-50"
            disabled={form?.pending}>
      {#if form?.pending}
        Entrando... <!-- texto cambia si estamos enviando -->
      {:else}
        Entrar
      {/if}
    </button>
  </form>
</main>
```

Algunos aspectos de este snippet: - Uso de `form` store de SvelteKit (actions) para obtener estado de envío (`pending`) y errores provenientes del servidor. - Atributos accesibles: el div de error tiene `role="alert"` para que lectores de pantalla anuncien el mensaje. - Clases Tailwind para estilo: `.input` y `.btn` podrían ser utilidades de shadcn-svelte o definidas en CSS, pero aquí ilustramos directamente. - `on:submit={() => invalidate('login')}` es una manera de forzar que, tras submit, si hay navegaciones, se invalide la data. (Este detalle técnico podría omitirse, pero se muestra cómo podríamos refrescar el estado tras login). - Código legible: nombres de variables `email`, `password`, etiquetas semánticas `<label>`, etc., facilitan la comprensión.

6.4 Ejemplo de Prueba Unitaria (Vitest)

Incluimos un ejemplo de prueba para el servicio de chat, demostrando cómo asegurar que la lógica funcione correctamente:

```
/// <reference types="vitest" />
import { describe, it, expect, vi } from 'vitest';
import * as chatService from '$lib/services/chat.service';
import { chatStore } from '$lib/stores/chat.store';
```

```
describe('Chat Service', () => {
  it('should not send message if socket is disconnected', () => {
    // Aseguramos que no hay socket conectado
    // (chatService.socket es interno; podríamos exponer un método
isConnected o simular desconexión)
    chatService.disconnectChatSocket();

    const result = chatService.sendMessage('conv123', 'Hola');
    expect(result).toBeNull();
    // Además, el store no debería tener mensajes añadidos
    let messages;
    chatStore.messages.subscribe(value => messages = value)();
    expect(messages).toEqual([]);
  });

  it('should add temp message and call socket.emit on sendMessage when
connected', () => {
    // Simulamos socket conectado usando un mock
    const fakeSocket = { emit: vi.fn(), connected: true };
    // Suplantamos el socket interno (asumiendo chatService expone
internamente o vía DI)
    (chatService as any).socket = fakeSocket;

    const tempId = chatService.sendMessage('conv123', 'Prueba de envío');
    expect(typeof tempId).toBe('string');
    expect(tempId?.startsWith('temp-')).toBe(true);
    // verificar que socket.emit fue llamado con los datos correctos
    expect(fakeSocket.emit).toHaveBeenCalledWith('message',
expect.objectContaining({
      conversationId: 'conv123',
      content: 'Prueba de envío'
    }));
    // verificar que el store contiene el mensaje temporal
    let lastMessage;
    chatStore.messages.subscribe(msgs => lastMessage = msgs[msgs.length - 1])
();
    expect(lastMessage.content).toBe('Prueba de envío');
    expect(lastMessage.status).toBe('sending');
  });
});
```

Esta prueba: - Usa Vitest para simular el comportamiento del socket. Al asignar `chatService.socket = fakeSocket` (usando casting a any para acceder, en un caso real se podría refactorizar para inyección de dependencias). - Envía un mensaje y comprueba: - Que devuelve un `tempId`. - Que `socket.emit` fue llamado con un objeto que contiene el contenido y convId esperado. - Que el store `chatStore.messages` recibió el mensaje con status 'sending'. - Este tipo de prueba asegura que la lógica de sendMessage funciona correctamente aislada de un entorno real.

Conclusión de la sección: Estos ejemplos demuestran nuestra intención de escribir código **claro, mantenible y auto-documentado**. Cada módulo, componente o función seguirá patrones similares: - Nombres descriptivos (en inglés técnico para el código, comentarios en español o bilingües si se

prefiere). - Comentarios útiles antes de bloques lógicos complejos o partes no obvias. - Formato consistente (Prettier lo asegurará en gran medida). - Estructura de carpetas y archivos intuitiva para que buscar algo en el proyecto sea sencillo.

Esta forma de codificar facilitará no solo la colaboración humana sino también que herramientas de IA puedan brindar ayuda contextual (por ejemplo, autocompletar funciones entendiendo nuestro código, o generar documentación automatizada).

7. Arquitectura para Mantenimiento, Escalabilidad y Testing

Una meta fundamental de este proyecto es que la arquitectura soporte la **evolución a largo plazo**. Cada decisión intenta minimizar la deuda técnica y hacer que añadir nuevas funcionalidades o módulos sea ordenado. Aquí resumimos las prácticas arquitectónicas y de convenciones que refuerzan la mantenibilidad y escalabilidad, así como el enfoque en testing continuo.

- **Modularidad por Dominio**: Como se vio en la estructura, agrupamos por módulo funcional (auth, chat, crm, etc.). Cada módulo nuevo se sumará siguiendo el mismo esquema:
- Carpeta de ruta para sus páginas,
- Stores específicos si requieren manejar estado global (ej. `crm.store.ts`),
- Servicios específicos para llamadas API (ej. `crm.service.ts`),

- Componentes UI específicos (ej. `CustomerCard.svelte` en `components/crm/`).
  Esta separación evita colisiones y hace más fácil que diferentes equipos o desarrolladores trabajen en paralelo en distintos módulos sin pisarse.

- **Convenciones de Nomenclatura**:

- Archivos Svelte de componentes en `PascalCase.svelte` (ej. `ChatWindow.svelte`), para distinguirlos de páginas (que van como +page.svelte) y de archivos de script.
- Archivos de stores/servicios en `camelCase` o `kebab-case` según preferencia, pero consistente. Por ejemplo, optamos por `.store.ts` sufijo para claridad.
- Variables y funciones en inglés, descriptivos: por ejemplo, usar `sendMessage()` en lugar de `send` a secas, `userProfile` en lugar de `data` para un objeto usuario. Esto reduce confusiones.
- Constantes en mayúsculas (ej. `MAX_FILE_SIZE_MB`).

- Comentarios JSDoc para exportaciones importantes (stores, servicios) explicando su propósito.

- **Separación de lógica y presentación**:

- Regla general: **componentes Svelte = presentación (vista)**, **services/stores = lógica de negocio/estado**. Por ejemplo, la validación de un formulario sencilla puede estar en el componente, pero la acción de hacer login está en auth.service. Esto hace que la lógica sea testeable sin UI y que la UI sea fácilmente reemplazable o modificable sin tocar la esencia.

- Eventualmente, podríamos extraer lógica más compleja a *Custom Hooks* (no hooks de React, sino funciones que encapsulan funcionalidad, p.ej., una función useChat() que suscribe a chatStore y deriva estados, etc.). En Svelte se suele hacer directamente en stores, pero es un patrón a considerar.

- **Preparado para Equipo y Colaboración**:

- Un beneficio de SvelteKit + nuestra estructura es que si en el futuro hay múltiples equipos (uno para chat, otro para CRM), pueden trabajar casi aisladamente en sus carpetas. El `lib/` común contendrá utilidades y componentes transversales (como la librería UI) que deben seguir convenios claros para no romper estilos.

- Documentación interna: Junto con este plan, se debería mantener un **docs/** en el repo con quizás guías breves de "Cómo añadir un módulo", "Cómo usar X componente", etc., que complementen la facilidad que ya da el código legible.

- **Escalabilidad Técnica**:

- La elección de SvelteKit nos permite escalar en términos de usuarios y carga gracias a SSR y buena performance. Si la base de usuarios crece, la aplicación puede migrar a un CDN/edge easily (usando adaptadores).
- Escalar funcionalmente: gracias a la modularidad, agregar nuevas páginas o flujos no afecta los existentes. Por ejemplo, implementar un **módulo de videollamada** en el futuro podría ser un directorio `calls/` con su propia sub-app; mientras no interfiera en global (salvo quizás en navigation menu).
- **Bundle splitting**: SvelteKit automáticamente divide el JavaScript por rutas. Esto significa que si un usuario solo usa Chat, no cargará el código de CRM hasta que navegue allí. Esto mantiene la app rápida incluso al añadir muchas features. Debemos sin embargo vigilar dependencias globales: por ejemplo, si agregamos una pesada (digamos, a charting library for campaigns) y la importamos global, romperíamos este beneficio. Mejor cargarla solo en las páginas necesarias (dynamic import si es muy pesada).

- **Lazy loading assets**: Además del código, cargaremos de forma diferida recursos como imágenes de iconos (a menos que usemos SVG inline) y posiblemente módulos de terceros. Ejemplo: el módulo de campañas quizás necesita Google Maps API; podemos asegurarnos de no cargarlo hasta entrar a campañas.

- **Testing desde el Día 1**:

- Ya configuramos un ambiente de testing, lo importante es la cultura: escribir tests **mientras** desarrollamos funcionalidades, no al final.
- Para cada store o service añadido, acompañarlo de tests unitarios. Esto actúa como documentación viva y evita regresiones. Por ejemplo, si cambiamos cómo funciona refreshToken, los tests fallarán si rompemos algo.
- Para cada componente complejo (ej. ChatWindow), crear tests de componente para verificar que renderiza estados importantes. Testing Library con jsdom nos permite simular casos: "dado X mensajes en store, el componente muestra X mensajes en el DOM".
- **Integración continua**: los tests correrán en CI, por lo que ningún código rompe será mergeado sin ser corregido.
- **Pruebas E2E**: A medida que integremos módulos, los tests end-to-end deben abarcar escenarios críticos: login -> enviar mensaje -> ver que otro usuario recibe (quizás simulando dos usuarios con 2 browser contexts en Playwright). Esto se puede hacer en pipeline o al menos en staging antes de release.

- Todo esto asegura que al escalar el sistema con nuevas funciones, podemos refactorizar con confianza, respaldados por las pruebas.

- **Documentación y Comentarios**:

- Fomentaremos que cada módulo nuevo venga con README técnico interno o al menos buenos comentarios. El código *"self-documented"* es la meta, pero cuando hay decisiones arquitectónicas (ej. "Se usa algoritmo X por razón Y"), dejar un comentario es valioso.
- Además, mantener actualizado el archivo CHANGELOG.md o notas de versiones, para registro histórico.

- **Mantenibilidad**:

- Code linting y formatting no solo son para estilo, previenen ciertos bugs (ej. el linter de Svelte nos recordará añadir `bind:this` en ciertos casos o cerrar etiquetas).
- Utilizar las actualizaciones de Svelte/SvelteKit: como proyecto moderno, habrá updates frecuentes. Planificar tiempo para version upgrades, y tener tests hace que actualizar (por ej. a SvelteKit 2.x) sea menos riesgoso.
- Monitorear performance con las herramientas de Svelte (Svelte DevTools) y auditorías (Lighthouse) de vez en cuando, para evitar degradaciones silenciosas.

En resumen, la arquitectura propuesta está pensada para **crecer limpiamente**. Cada pieza está desacoplada en lo posible, probada y documentada. Esto reduce la probabilidad de errores al extender el sistema y facilita localizar bugs (pues se sabe qué responsabilidad tiene cada módulo). Todo esto se traduce en un menor costo de mantenimiento a largo plazo.

8. Seguridad, Performance, Accesibilidad y Otras Consideraciones Transversales

Además de la funcionalidad, un frontend de clase mundial debe destacar en **seguridad**, **rendimiento**, **accesibilidad (a11y)** y buenas prácticas generales (caché, monitoreo, etc.). En UTalk, prestaremos atención a estos aspectos desde el inicio:

8.1 Seguridad Frontend
- **Contenido Activo & XSS**: Como se mencionó en el chat, evitaremos usar `{@html}` con contenido de usuarios. Toda inserción de datos de usuarios en la interfaz será escapada automáticamente por Svelte (que es el comportamiento por defecto al usar `{variable}`). Solo usaremos HTML sin escape si es contenido nuestro o previamente filtrado. - **Sanitizar Inputs**: Aunque la validación principal es del lado servidor, podemos sanitizar o validar longitud de inputs en el cliente para usabilidad. Por ejemplo, recortar espacios excesivos en nombre de usuario, etc., antes de enviar. - **Protección contra Inyecciones**: En contexto frontend, significa no construir manualmente HTML o queries, así que poco aplica. Pero evitaremos usar eval o funciones peligrosas. SvelteKit de base no expone esas vectores a menos que se usen mal. - **Uso de HttpOnly Cookies**: Reiterando, esta es una protección fuerte contra XSS para tokens [6]. Adicionalmente, configurar la cookie con `Secure` y `SameSite`. Si la app es single-domain (frontend y backend en el mismo dominio), `SameSite=Lax` es suficiente para evitar envíos cross-site en la mayoría de contextos (formularios top-level están cubiertos). Si hay subdominios, se ajustará `SameSite=None` con `Secure` y se implementará un token CSRF. - **Headers de Seguridad**: Al desplegar, asegurarse de enviar headers como: - `Content-Security-Policy` que restrinja cargas de recursos a fuentes conocidas (evitar que un XSS inyectado cargue script externo). - `X-Frame-Options: DENY` para prevenir clickjacking a la app en iframes. - `Strict-Transport-Security` para forzar HTTPS. - SvelteKit permite configurar headers en handle or via adapter config. Lo haremos en su momento de deploy. - **Autorización en UI**: Aunque las reglas de acceso son validadas en backend, en frontend no mostraremos opciones que el usuario no debería usar. Por ejemplo, si un usuario es rol "agente" y no debe ver módulo "Administración", no renderizar ese ítem de menú. Esto se

logra consultando `authStore.user.role` al construir la navegación. - **Escape de datos en URLs**: Siempre que generemos URLs con datos de usuario (por ejemplo, query params con search terms), usar funciones de encode (`encodeURIComponent`) para prevenir inyección de caracteres especiales. - **Bibliotecas actualizadas**: Mantener dependencias npm actualizadas para recibir parches de seguridad (esto lo monitoreará `npm audit` y herramientas como Dependabot). - **Deshabilitar funciones peligrosas en producción**: Por ejemplo, asegurarse de no dejar logger activado o endpoints de prueba. SvelteKit puede tree-shake imports de dev (podemos usar `if (import.meta.env.DEV) console.log(...)` para que se elimine en prod). - **Captcha/2FA**: Si en un futuro se requiere mayor seguridad en login (evitar bots), la arquitectura admite añadir un reCAPTCHA en el formulario de login, o un segundo factor. Esto no se implementa de inicio, pero la base de código es compatible.

8.2 Performance

- **SSR Caching**: SvelteKit SSR puede beneficiarse de caché en ciertas páginas. En UTalk, la mayoría de páginas son dinámicas tras login (no cacheables globalmente), pero podríamos cachéar respuestas estáticas (como los assets) fuertemente. - Configurar el servidor (o Vercel) para servir assets estáticos con `Cache-Control: max-age=31536000` (1 año, con fingerprint en nombre de archivo para invalidar al actualizar). - Para datos de chat, no cacheamos ya que son personalizadas y tiempo real. - **Code Splitting & Async**: Ya discutido, cada módulo por separado. Además, podemos marcar algunos componentes como dinámicos: - Por ejemplo, un componente pesado de gráficos en campañas, cargarlo con `import(...)` al montarlo la primera vez. - Si hay rutas que no requieren SSR (como quizás el chat mismo si preferimos cargar cliente), se pueden marcar como SPA-only, pero en general SSR ayuda en inicial. - **Prefetching**: SvelteKit tiene `<a use:prefetch>` para cargar de fondo la página siguiente cuando el usuario hace hover en un link. Podemos usar esto en elementos de navegación (ej. al ver la lista de conversaciones, prefetch la página de chat de la primera conversación). - También, si hay datos que podemos pre-obtener, por ejemplo la lista de conversaciones al cargar la app, se podría cargar en paralelo con la página chat (depende de diseño: Slack carga lista de chats y contenido juntos). - **Optimización de Imágenes**: Si usuarios suben avatares o imágenes, implementar: - **Thumbnais**: Mostrar versiones reducidas en listados (ej. avatar 40x40). Si backend no lo da, usar CSS `width/height` para al menos no renderizarlas enormes. A futuro, integrar un servicio de imágenes responsive. - **WebP**: Navegadores modernos soportan WebP/AVIF. Si tenemos control, convertir imágenes a esos formatos en backend y servirlos. En front, usar `<picture>` para proveer varias resoluciones/formats. - **Lazy Loading**: Ya mencionado `loading="lazy"` en <img> ensures offscreen images (like earlier messages above viewport) won't load until needed. - **Minificar y Compress**: Vite/SvelteKit produce código minificado. Adicionalmente, en el servidor habilitar `gzip` / `brotli` compression for HTML/JS/CSS. Esto reduce tiempos de transferencia. - **PWA & Offline**: Un paso adicional: convertir UTalk en una *Progressive Web App*. - Instalar un Service Worker (SvelteKit tiene soporte experimental con `pwa` options o we can write one) para caching static assets y maybe last messages for offline read. - Permitir a la app funcionar si pierde conexión: ver historial (cacheado), escribir mensajes en cola (los enviará al reconectar). Esta funcionalidad offline es avanzada pero alcanzable con nuestra architecture (store de pending messages etc.). - Agregar manifest.json e iconos para instalable en móvil. Esto puede considerarse en un milestone posterior (Fase 5+). - **Monitoring Performance**: Integrar alguna analítica de performance (por ej. Google Analytics puede dar Core Web Vitals, o usar Sentry Performance Monitoring). Al menos usar el API de Performance del navegador para medir tiempos de carga en diferentes condiciones (contrastar SSR vs CSR). - Podemos configurar un profile en Lighthouse (por ej. as part of CI, run a Lighthouse check on a deployed URL to catch regressions in perf).

8.3 Accesibilidad (a11y)

Desde el diseño: - **Semantic HTML**: Usar elementos correctos (formularios con labels, botones genuinos `<button>` en vez de divs clicables, headings jerárquicos `h1, h2, ...` según la sección). Esto ayuda a lectores de pantalla a interpretar la app. - **Focus Management**: En un SPA, cuando se

navega programáticamente (ej. tras login redirigir a `/chat`), asegurarse de mover el foco al inicio de la nueva página o a un heading principal. SvelteKit por defecto maneja scroll, pero podemos añadir `tabindex="-1"` en el título y focus en onMount. - **Contraste de color**: Configurar Tailwind con colores que cumplan WCAG AA. Texto sobre fondos debe tener suficiente contraste. Probar combinaciones (existen plugins de tailwind for contrast or just manual testing). - **Modo Oscuro**: Si se soporta (shadcn-svelte facilita toggling), asegurarse también de contrastes en dark mode. - **Teclado**: Toda funcionalidad debe ser accesible vía teclado: - Navegación entre elementos con `Tab` debe seguir orden lógico. Use `tabindex` if needed for custom elements. - Atajos: podríamos introducir atajos (ej. Ctrl+K to search, etc. in future). - Componentes complejos (dropdowns, modals) deben manejar foco interno. Los de shadcn-svelte suelen venir con accesibilidad incorporada (e.g., Dialog traps focus, etc.). - **Aria y roles**: Añadir `aria-label` o `aria-describedby` en iconos o botones ambiguos. Por ejemplo, el botón de enviar (icono de paper plane) debería tener `aria-label="Enviar mensaje"`. - Roles ya mencionados, e.g. alert for error messages. - **Testing accesibilidad**: Podemos integrar **axe-core** (herramienta de análisis a11y) en dev/testing. E.g., en Playwright tests, correr `await expect(page).toHaveNoAccessibilityIssues()` con Axe. - **Localización**: A futuro, si UTalk se ofrecerá en múltiples idiomas, desde ya escribir textos en un archivo de constantes o usar i18n libs. Pero inicialmente en español es fine; sin embargo, estructurar para traducción más tarde es parte de accesibilidad global (no todos los usuarios hablan español, eventualmente).

8.4 Manejabilidad, Monitoreo y Logs
- **Logs de Errores**: Integrar Sentry (o similar) para capturar errores runtime en producción. Sentry puede agrupar errores, darnos stack traces. SvelteKit integration is straightforward (init in hooks, capture in handleError). Establecerlo antes de ir a producción. - **User Feedback en Errores**: Cuando ocurra un error no anticipado (ej. fallo en fetch, componente que lanza excepción): - SvelteKit mostrará +error.svelte global. Debemos personalizarlo: un mensaje "Oops, algo salió mal" y un botón para recargar o volver a inicio. - Podemos loguear el error a Sentry en ese momento (SvelteKit docs show how to use `handleError` hook). - **Monitoreo de Uso**: Podríamos querer saber cuántos usuarios usan chat concurrentemente, etc. Para eso, quizás el backend ya monitoriza. En front podríamos tener Google Analytics or simple metrics (page view, etc.), pero dado que es app interna mayormente, no es prioritario. Aun así, es bueno tener la opción abierta. - **Notificaciones Push**: Como parte de UX pero también tecnica: - Web Push: implementar en service worker, para notificar nuevos mensajes cuando el usuario esté offline. Requiere backend push service. Dejamos planificado: arquitectura actual (SvelteKit + service worker) lo soportaría. - In-App notifications: ya cubierto con realtime events en chat.

8.5 Caché y Almacenamiento
- **Caché de Datos**: Podemos usar SvelteKit Load functions with `depends` and `invalidate` to cache certain data on client. Por ejemplo, la lista de contactos o conversaciones podría cachéarse en session storage to avoid re-fetch on every nav. - Or use a store as cache: e.g., `contactsStore` loads once. - **Local Storage**: Para pequeñas cosas no sensibles: e.g., guardar preferencia de tema (oscuro/claro), recordar el último estado de la UI (última conversación abierta) para restaurar al volver. - For chat, maybe store draft messages per conversation in localStorage to persist them if user navigates away and back. - **IndexedDB**: Si offline and caching whole messages, using IndexedDB for storing messages could be considered. But to avoid over-engineering at start, keep it simple. If needed, use something like `localForage` or `Dexie` to abstract IDB.

- **Bundle Size**: Monitor via build stats. Keep an eye that the main bundle remains small (Svelte is good here). If using third-party libs, import only what's needed. If shadcn-svelte, tree-shaken because we only add components we use.

En esencia, estas consideraciones aseguran que la aplicación no solo **funcione**, sino que lo haga de forma **segura, rápida y usable para todos los usuarios**. Abordarlas desde el comienzo nos evita tener

que refactorizar por problemas de seguridad o performance más adelante, y nos acerca a un producto de calidad profesional.

9. Preparación para Módulos Futuros (CRM, Configuración, Campañas, Base de Conocimiento, etc.)

Aunque la prioridad inicial es login y chat, la arquitectura debe dejar el **camino despejado** para añadir módulos importantes en UTalk. A continuación, se describen lineamientos para extender hacia esos módulos, de modo que cuando llegue el momento, la integración sea fluida:

- **Navegación Común y Layout**: Probablemente, tras el login, el usuario ingrese a una vista general (dashboard o directamente chat). Pero cuando existan múltiples módulos (CRM, campañas...), necesitaremos una **barra de navegación lateral** o menú principal para moverse entre ellos.
- Diseñar desde temprano un `<MainLayout>` (usado en `routes/+layout.svelte`) que contenga esa navegación condicional: por ejemplo, un sidebar colapsable con opciones: Chat, CRM, Settings, etc. Inicialmente, en Chat MVP, ese sidebar podría tener solo Chat (seleccionado) y quizás placeholders/grayed out for others or simply not visible. Pero tenerlo en la estructura significa que añadir opciones es trivial.
- Cada módulo cargaría en el `<slot>` de ese layout común. Así se mantiene un look & feel consistente (no que CRM tenga otra plantilla distinta).
- Para usuarios con roles distintos, podemos controlar visibilidad: p.ej., modulo "Admin" visible solo para admins (basado en authStore.role).

- Incluir iconos y nombres claros para cada sección, de preferencia usando componentes de icono (heroicons via shadcn maybe).

- **Stores por Módulo**:

- Ya contemplamos authStore, chatStore. Similarmente, podríamos añadir:
  - `crmStore` para estado compartido en CRM (ej. filtros aplicados, cliente seleccionado, etc.).
  - `settingsStore` para valores de configuración temporales (aunque muchas settings serán individuales y no necesitan store, algunas globales como "darkMode" puede ser store).
  - `campaignsStore` para datos de campañas (ej. campaña actual, resultados resumen).
  - `knowledgeStore` para la base de conocimiento (ej. artículo actual, categoría seleccionada).

- Estos stores vivirán en `lib/stores` y se documentarán igual que los existentes. Además, podrán reusar estructuras: por ejemplo, un `ListStore` genérico para paginación de listas (CRM list of clients, etc.) podría ser creado para reutilizar patrones.

- **Services por Módulo**:

- Anticipamos archivos en `lib/services` como `crm.service.ts`, `campaign.service.ts`, etc. Allí encapsularemos llamadas API de esos módulos. Esto mantiene la separación de concerns.
- P. ej., `crm.service.fetchLeads()` -> GET `/crm/leads`, `campaignService.launchCampaign(data)` -> POST `/campaigns`.

• Ya que la config base (URL, token handling) está en `api.ts`, todos los servicios lo comparten.

• Reutilizar: Si varios módulos requieren funcionalidad similar (ej. subir archivos se usa en chat y knowledge base), centralizarlo en `file.service.ts` como hemos hecho.

• **Lazy Loading Módulos**:

• Con SvelteKit file routing, si un usuario nunca visita "Campaigns", el código de campañas no se carga. Sin embargo, al build, todo existe. Asegurarse de no tener `import campaignService` en chat por error, etc.
• Si anticipamos que algunos usuarios usarán solo una parte de la app (ej. agentes solo chat y CRM, marketing solo campañas), esta separación por rutas ya nos da performance.

• En caso de una funcionalidad súper pesada, podríamos usar `import()` al entrar en la página. Pero SvelteKit ya maneja splitting por route, así que preferimos confiar en eso.

• **Design System Consistente**:

• Con shadcn-svelte, definiremos estilos base que todos los módulos compartirán (tipografías, colores, spacing). Al añadir nuevos módulos, seguirán las mismas guías. Esto garantiza cohesión visual.
• También crear componentes reusables desde ya pensando en su uso global: por ej., una tabla paginada componente que se usará tanto en CRM (lista de clientes) como en base de conocimiento (lista de artículos). Mejor implementarla de forma genérica en `components/ui/Table.svelte` para no duplicar.

• Notar: shadcn-svelte tiene componentes Data Table, etc., que podemos aprovechar.

• **Permisos y Roles**:

• Si los módulos futuros tienen control de acceso (muy probable, ej. solo ciertos roles pueden ver campañas), necesitamos incorporarlo.
• Esto se puede lograr:
    ◦ Lado servidor: en hook, si user.role no tiene permiso a ruta X, hacer redirect 403 or a "Not authorized" page.
    ◦ Lado cliente: no mostrar secciones no autorizadas en la nav.
• Para soportar esto fácilmente, definiremos quizá un mapa de rutas -> roles permitidos, en una constante. Y el hook global puede checar eso similar a cómo protegemos login.

• Por ahora, si solo hay un rol (o todos ven chat), no complicamos, pero tenerlo en mente.

• **Escalabilidad de Datos**:

• Módulos como CRM o campañas pueden manejar muchos registros, tablas, etc. Nuestra estructura con servicios y stores soporta manejar paginación, filtros, etc. Podríamos integrar librerías si necesario (ex: for charts in campaigns, a charting lib; for rich text in knowledge base, a text editor Svelte component).

• Nada en la arquitectura actual impide integrar librerías cuando se necesiten, dado que tenemos segregación (podemos import Quill editor in knowledge pages only, etc.).

• **Consistencia en Interacción**:

- Asegurar que por ejemplo, las notificaciones y toasts de error se comporten igual en todos módulos. Posiblemente implementar un `ui/Toast.svelte` central y un `toastStore` global que cualquier módulo use para disparar mensajes al usuario. Así evitamos que cada módulo cree su propio mecanismo.
- Parecido con modales, confirm dialogs, etc., tenerlos globales (shadcn-svelte provee `<Dialog>` que se puede usar en cualquier parte pero necesitamos un contenedor en root layout).

- **Componente Knowledge Base**:

- Podría requerir un viewer de artículos con formato (markdown?). Planear que sea un componente que podemos renderizar con sanitized HTML (posible uso de MDsvex or so, pero en runtime tal vez).
- Este es un ejemplo de "no sabemos los detalles aún pero sabemos que nuestra stack (Svelte + maybe an MD parser) es capaz".

- **Microservicios front?**:

- Probablemente no necesario: SvelteKit es monolítico pero puede manejar secciones muy distintas. A menos que la app sea gigantesca, mantendremos un solo repo/frontend.
- Pero si decidieran separar (ej. una app independiente para Knowledge base?), es posible más adelante extraer. Por ahora, monorepo monolito es más simple y las optimizaciones de code-splitting nos bastan.

- **Estado global vs estado por módulo**:

- Minimizar estado global compartido innecesariamente. AuthStore es global por naturaleza. ChatStore es global para chat, pero CRM quizás no necesita uno global a toda app (depende, si CRM moderate complexity, just fetch on component).
- Sin embargo, a veces querremos compartir cosas: ej. una notificación global "tienes X mensajes no leídos" visible en el header, requiere que chatStore sea global. O "tienes N tareas CRM pendientes" requiere CRMStore global.
- Podemos en esos casos exponer pequeños derivados (computed values) en esos stores para que la nav los muestre.
- Así la nav (que es global) depende de multiplés stores modulales, pero cada store sigue encapsulado.

- **Fases de Entrega**:

- Planificar que tras chat, vendrán releases incrementales para otros módulos. Cada módulo podría ser desarrollado en paralelo después de chat. La base construida (auth, UI library, config, CI) les sirve a todos.
- Se puede crear ramas independientes para cada módulo y luego integrarlas, gracias a modularidad, las fusiones serán más fáciles (menos puntos de conflicto).

En resumen, la arquitectura actual actúa como un **esqueleto** robusto donde es fácil conectar nuevos **miembros** (módulos) sin alterar lo ya hecho. Siempre siguiendo las mismas pautas de calidad: componentes reutilizables, servicios para API, stores para estado, etc. Cuando llegue el momento de

añadir CRM o cualquier otro, gran parte del trabajo será simplemente **replicar el patrón**: crear rutas, UI y lógica específica, reutilizando lo común (auth, UI kit, etc.).

10. Hitos (Milestones) y Definición de "Done" por Fase

Para gestionar el proyecto de manera eficaz, lo dividiremos en **fases** desde la configuración inicial hasta el lanzamiento del chat en producción. Cada fase tiene objetivos claros y criterios de finalización ("Definition of Done") para saber cuándo podemos pasar al siguiente paso.

A continuación se definen las fases 0 a 5:

- **Fase 0: Setup Inicial del Proyecto**
  Objetivo: Sentar las bases técnicas del frontend.
  Tareas principales:
- Inicializar proyecto SvelteKit con TypeScript.
- Configurar Tailwind CSS y agregar 2-3 componentes base de shadcn-svelte (ej. Button, Input, Dialog) para verificar la integración.
- Implementar ESLint, Prettier, Husky hooks (pre-commit, pre-push) y asegurar que funcionan.
- Configurar CI con pipeline de build y test de ejemplo.
- Establecer `authStore` y `chatStore` vacíos con estructura prevista.
- (Opcional) Página de bienvenida temporal que diga "UTalk Frontend" para probar deploy. **Done Criteria:**
- Repositorio con estructura base y configuraciones committeadas.
- `npm run dev` levanta la app sin errores y con Tailwind funcionando.
- Linter no muestra errores; commit hook impide commits con código mal formateado.
- Pipeline CI corre en cada push y pasa las tareas (lint/test/build) en un commit de prueba.
- Documentación básica (README) explicando cómo correr el proyecto localmente y describiendo la estructura (puede referenciar secciones de este plan).

- Equipo sincronizado: todos pueden clonar repo, instalar y ver el "Hello World" corriendo.

- **Fase 1: Implementación del Sistema de Login**
  Objetivo: Desarrollar completamente la funcionalidad de autenticación de usuarios.
  Tareas principales:

- Crear página `/login` con formulario de ingreso (email & password) y validaciones de frontend.
- Implementar `authService.login` llamando al backend real (si disponible; sino mockear respuesta).
- Manejar token JWT: guardar en cookie HttpOnly o localStorage según decidido (preferencia cookie).
- Configurar hook `hooks.server.ts` para proteger rutas internas: redirigir a /login si no autenticado.
- UI de errores de login (mensaje si credenciales inválidas).
- Implementar logout: por ahora quizás como un botón temporal en alguna parte (lo integráremos en chat UI después).
- Asegurar persistencia de sesión al refrescar (ver que hook lee cookie y repuebla `authStore`).
- Pruebas: unit tests para authService (simulando casos success/fail), componente login form (que muestra errores), y idealmente un test e2e: login con credenciales de prueba navega al area interna. **Done Criteria:**

- Un usuario puede ingresar sus credenciales en /login y, con un backend correcto, obtener acceso a la app (redirección a página interna /chat o dashboard).
- Si introduce datos erróneos, ve un mensaje de error claro.
- No puede navegar a rutas internas (ej. /chat) sin estar autenticado (es redirigido de vuelta a /login).
- El token de sesión se almacena de forma segura; al recargar la página, sigue autenticado (hasta expirar el token).
- Logout elimina la sesión y redirige a /login.
- Pruebas automatizadas cubren los casos de login exitoso, error de login, y protección de ruta (p.ej., un test trata de acceder /chat sin login y espera ser a /login).

- Código revisado y cumpliendo estándares (review de colegas o al menos pasaron todos linters/format).

- **Fase 2: Estructura Base del Módulo de Chat**
  Objetivo: Montar la interfaz principal de chat y establecer la conexión en tiempo real (aunque inicialmente sin todas las funcionalidades avanzadas).
  Tareas principales:

- Crear página `/chat` que muestre la estructura: lista de conversaciones a la izquierda (dummy data por ahora), ventana de mensajes a la derecha.
- Integrar Socket.io cliente: abrir conexión al entrar a /chat (por ejemplo en `onMount` del componente chat page o via `chatService.connectChatSocket` llamado en `+page.svelte`).
- Mostrar una conversación dummy: por ejemplo, "Selecciona un chat a la izquierda" si ninguno seleccionado, o abrir una por defecto.
- Implementar `ChatInput` componente con textarea y botón enviar (no funcional aún o solo simulado).
- Habilitar envío de mensaje de texto simple:
    - Por ahora, si backend socket disponible, probar envío real. Sino, simular mensaje retornando (echo).
    - Añadir mensaje al chat UI inmediatamente (aunque sea sin persistir).
- Manejar recepción de mensajes:
    - Simular recepciones (por ejemplo, usar socket.on('message') y push a UI).
    - Probar con 2 ventanas del app (si backend real, 2 usuarios).
- Estado de conexión: indicar en UI simple (un icono conectado/desconectado).
- No implementar adjuntos ni detalles de status todavía, pero dejar espacio en UI (por ejemplo, icono de clip inactivo).
- Pruebas: unit test de chatStore (agregar mensaje), componente ChatInput (que al hacer submit llama servicio), quizás e2e: dos usuarios chat (si ambiente lo permite). **Done Criteria:**
- Pantalla de chat accesible tras login, mostrando layout básico (lista de chats, ventana de mensajes).
- Conexión WebSocket establecida (verificado por logs o por icono "online").
- Usuario puede enviar un mensaje de texto y verlo aparecer en la ventana (ya sea confirmado por el servidor o inmediatamente local).
- Si hay otro usuario (o simulación), los mensajes recibidos aparecen en la lista en tiempo real.
- La aplicación maneja desconexión simple: si se apaga el server de chat, reconecta (probado simulando).
- Ningún error JS mayor en consola durante esas operaciones.
- El código de chat está estructurado (servicios, stores, componentes) aunque falten features, pero nada improvisado en malos lugares.

• Tests básicos pasan: e.g., chatStore adding message, sendMessage returns tempId, etc.

• **Fase 3: Funcionalidades Avanzadas del Chat**
Objetivo: Completar el módulo de chat con todas las características: multimedia, estados, notificaciones in-app, etc.
Tareas principales:

• **Multimedia**: Implementar adjuntar imágenes:
  ◦ Soportar vista previa al adjuntar.
  ◦ Subir archivo a backend y manejar la respuesta.
  ◦ Enviar mensaje con imagen (mostrar thumbnail en chat).
  ◦ Recibir y mostrar imagen enviada por otros.
• **Estados de mensajes**: Integrar confirmación de entrega/lectura:
  ◦ Mostrar check(s) en mensajes enviados. (Requiere que backend emita ack de recibido y quizá de leído).
  ◦ Actualizar UI cuando usuario destino lee mensajes (por simplicidad, si chat uno a uno, marcar todos previos como leídos).
• **Typing indicator**: Enviar evento 'typing' cuando usuario escribe, mostrar "X está escribiendo..." cuando recibimos uno.
• **Manejo de errores**:
  ◦ Si enviar mensaje falla (por no conexión en ese instante), marcar mensaje como error y permitir reintentar (e.g., pulsando un botón reintentar reenvía).
  ◦ Si subir archivo falla, mostrar error y opción de reintentar.
• **Notificaciones**:
  ◦ In-app: si usuario está en otra conversación o minimizado el chat, y llega mensaje nuevo en otra conv, mostrar un toast o resaltar la conversación en la lista (ej. en negrita con número de mensajes nuevos).
  ◦ Push notifications: (opcional) Si factible, integrar el API Notifications: pedir permiso al usuario, y si está en segundo plano, disparar notificación del navegador en nuevos mensajes. Esto puede posponerse, pero dejarlo listo conceptualmente.
• **Performance enhancements**:
  ◦ Paginación de historial: permitir cargar más mensajes antiguos con scroll.
  ◦ Virtualize list if needed (test with many messages).
  ◦ Ensure UI fluid with large text or many quick messages (maybe throttle scroll).
• **UI refinements**:
  ◦ Estilos pulidos para burbujas de mensajes (diferente color emisor/receptor, esquinas redondeadas, etc.).
  ◦ Mostrar nombre y foto (avatar) del otro usuario en top del chat, y pequeños avatares junto a sus mensajes (en grupo chats, etc., identificar emisor).
  ◦ Responsive design: verificar que en pantallas móviles la UI se adapta (quizá lista de chats se oculta tras un botón de menú, etc.).
  ◦ Asegurar que componentes como Modal (ej. preview imagen) funcionan y son accesibles.
• **Testing**:
  ◦ Casos de unidad: sending image flows (podríamos simular fileService).
  ◦ E2E: dos usuarios intercambiando mensajes, incluyendo imagen, verificar que el flujo completo corre sin errores y con UI actualizada. **Done Criteria:**
• El chat ofrece una experiencia completa: los usuarios pueden chatear fluidamente con texto e imágenes, con feedback claro de envío y lectura.
• Cualquier mensaje enviado llega a destino (en tests manuales con dos navegadores, confirmamos).

- Si el usuario A está en chat con B, y A envía un mensaje, B ve "mensaje nuevo" notificación si no tiene esa conv abierta.
- El sistema se recupera de desconexiones: se reconecta solo, y los mensajes enviados durante caída no se pierden (se envían al reconectar).
- La interfaz es agradable y estable en desktop y móvil (valide manual con dev tools mobile view).
- No quedan funcionalidades "pendientes" del listado de requisitos del chat (envío, recepción, multimedia, reconexión, notificaciones, seguridad de sanitización).
- Código revisado: esta es una fase grande, probablemente necesita revisión por pares o QA interno. Criterio done incluye "bugs conocidos del chat corregidos".

- Cobertura de pruebas: asegurar que la mayoría de funciones del chat tengan al menos tests básicos (no es necesario 100% coverage, pero las partes críticas como send, receive, file upload, ack handling sí).

- **Fase 4: Polishing, Performance Tuning y Preparación Producción**
  Objetivo: Antes del lanzamiento del módulo chat, abordar temas transversales: optimizaciones, seguridad final, accesibilidad y pruebas integrales.
  Tareas principales:

- **Auditoría de Performance**: Correr Lighthouse o WebPageTest en un build de producción:
  - Mejorar métricas de LCP, TTI si hay problemas (p.ej., dividir código si una página inicial está lenta).
  - Ver tamaño de bundle, eliminar dependencias innecesarias.
  - Comprobar que imágenes están optimizadas (tal vez configurar algo como `@sveltejs/adapter-auto` con compress).
- **Auditoría de Seguridad**:
  - Revisar que no queden tokens en localStorage (si migramos a cookies).
  - Probar flujos anómalos: XSS injection (mandando `<script>` en un mensaje, ver que aparece escapado, no ejecuta).
  - CSRF: asegurar endpoints críticos (login) no vulnerables (si cookies, quizá agregar token CSRF).
  - Revisar console no muestre warnings de libs vulnerables.
  - Actualizar todas las libs a última versión estable.
- **Accesibilidad**:
  - Pasar herramienta Axe y manual checks:
  - Todos los íconos tienen alt o aria-label.
  - Contrastes cumplen (ajustar Tailwind config si alguno no).
  - Navegación por tabulador recorre lógicamente. Añadir `skip to content` link si necesario al top.
  - Etiquetas ARIA donde falten.
  - Involucrar si es posible a alguien con lector pantalla para un breve test (o usar Narrator/VoiceOver nosotros).
- **Testing final**:
  - Escribir tests E2E de escenarios integrales:
  - Login -> enviar mensaje -> logout (flujo completo).
  - Reconexión: simular apagar socket server (podemos simular en dev if possible) y ver que reconecta.
  - Subir archivo pesado (simular tamaño borderline) -> ver manejo UI.
  - Pruebas multi-navegador: con Playwright, abrir 2 contexts (dos users) chateando reciprocamente, validar que ambos ven mensajes de cada uno.
  - Test de regresión sobre login features (nada roto al integrar con chat).

- **Documentation & Handoff**:
    - Escribir documentación de uso breve: quizá una página markdown con "User Guide" para testers/QA.
    - Completar/elaborar README y remove any "WIP" notes.
    - Plan de monitoreo: configurar Sentry DSN in env, ensure errors will be captured in prod.
- **Infrastructure**:
    - Preparar el entorno de producción:
    - Build final optimizado.
    - Configurar dominio, SSL (if self-host).
    - Variables de entorno de prod (API URL real, etc.).
    - Hacer un despliegue staging y una ronda de prueba QA formal. Arreglar cualquier bug.

    Done Criteria:
- Lighthouse score (Performance >90, A11y ~100, Best Practices/SEO as applicable).
- No errores de accesibilidad obvios (Axe con cero violaciones críticas).
- La aplicación se siente fluida, cargando rápidamente y sin memory leaks (monitoreado en devtools performance).
- Todos los tests (unit, integration, e2e) pasan exitosamente en CI.
- Aprobación de QA: se lista un conjunto de casos de prueba manuales todos marcados como OK.
- Documentación actualizada y entregada (incluyendo este documento plan si se mantiene).
- Sentry (u otra herramienta) está detectando errores en staging y ninguno mayor aparece tras fixes.

- Equipo de backend/devops está alineado para el release (sin pendientes de integración).

- **Fase 5: Lanzamiento del Chat en Producción**

  Objetivo: Desplegar la aplicación con el módulo de chat funcional a los usuarios finales, y monitorizar su funcionamiento.

  Tareas principales:

- **Despliegue en Producción**: Realizar el deployment final en el entorno producción (puede involucrar pipelines o pasos manuales aprobados).
- **Verificación post-deploy**: El equipo verifica que:
    - La app está accesible para usuarios reales.
    - Login funciona con cuentas reales.
    - Chat envía/recibe en real (tal vez probar con usuarios de prueba).
    - No hay errores en consola ni en network (200 OK en recursos).
- **Monitorización activa**: Establecer que en la primera semana:
    - Vigilar los logs de Sentry/monitor: ver si aparecen errores inesperados.
    - Vigilar performance: a través de GA o datadog rum si implementado, ver tiempos de carga, etc.
    - Feedback de usuarios: recopilar cualquier reporte de bug o queja y priorizar hotfix si crítico.
- **Soporte**: Asegurar que el equipo esté disponible para responder a incidentes (por ejemplo, si chat se cae, tener rollback plan).
- **Retrospectiva**: Reunirse para analizar el éxito de la release, anotar mejoras para próximos módulos (por ej. "Para CRM haremos X distinto."). **Done Criteria:**
- Usuarios finales están usando el chat en producción sin problemas graves.
- Métricas: % de logins exitosos, mensajes enviados, etc., confirman que la funcionalidad cumple (si se miden).
- No hay bugs severos abiertos post-lanzamiento; cualquier issue menor está agendado para corregir pero no impide uso.

- Se cumple con los requerimientos iniciales del producto para esta entrega (stakeholders están satisfechos).
- Se cierra formalmente el alcance de esta fase y se da luz verde para iniciar el siguiente módulo (ej. CRM) utilizando la base establecida.

Con estos hitos bien definidos, el proyecto puede avanzar de forma **iterativa**, entregando valor en cada fase y asegurando calidad mediante criterios de Done estrictos. Este plan maestro proporciona la guía integral para construir el frontend de UTalk con enfoque en excelencia técnica y preparación para un crecimiento sostenido. Cada miembro del equipo (y incluso asistentes de IA) puede referenciar este documento para entender la visión global y los detalles de implementación a seguir.

¡Manos a la obra para construir UTalk Frontend!

---

[1] Svelte Reviewed: A Masterclass on Empowerment - DEV Community
https://dev.to/somedood/svelte-reviewed-a-masterclass-on-empowerment-2544

[2] [3] [4] [5] Introduction - shadcn-svelte
https://shadcn-svelte.com/docs

[6] [7] [8] Protecting sveltekit routes from unauthenticated users - DEV Community
https://dev.to/thiteago/protecting-sveltekit-routes-from-unauthenticated-users-nb9