



29-1-2021

Chat-React-Firebase

Boxitas Applyment



Victor Artola Romero
jenviktor13@gmail.com

Contenido

<i>Solución para Chat-React-Firebase:</i>	1
Arquitectura y Tecnologías Utilizadas:	1
<i>Algunos Tipos de Arquitectura de aplicaciones de mensajería</i>	4
<i>Arquitectura impulsada por eventos + Simplicidad + Iteratibilidad</i>	4
Superar el límite de # de usuarios por sala de chat.....	6
Arquitectura de base de datos para la escalabilidad y el rendimiento de la aplicación	7
Consideraciones de ingeniería:	8
<i>Arquitecturas de chat basadas en plantillas: Firebase y Layer</i>	10
<i>Arquitecturas de chat descentralizadas</i>	12
<i>Arquitecturas frontend para crear una aplicación de mensajería</i>	15
Arquitectura frontend de la aplicación de chat basada en Angular	15
Deuda técnica con aplicaciones de chat basadas en Angular	16
Cómo reducir la deuda técnica.....	16
Deuda de documentación inicial.....	17
Arquitectura frontend de la aplicación de chat basada en React	17
<i>Cómo hacer una aplicación de mensajería: características y selección de plataforma</i>	19
Videollamada	19
Sincronización automática de contactos.....	20
Indicador de escritura en una aplicación de chat.....	21
Uso compartido de archivos grandes.....	22
En la cámara de la aplicación.....	22
Función de historias de tipo Instagram	23
Encriptado de fin a fin	23
Notificación de inserción.....	23
Actualizaciones de la pantalla de chat en tiempo real	24
<i>Arquitecturas basadas en plataformas estándar</i>	24

Solución para Chat-React-Firebase:

<https://gitlab.com/jenviktor13/chatreact>

Arquitectura y Tecnologías Utilizadas:

Frontend: [React.js](#)

Backend: [Google Firebase](#)

Árbol

- ❖ Components Folder
 - Pages Folder
 - Index.js
 - Signup.js
 - Layouts
 - Layout.js
 - Login.js
 - Chat.js
 - Message.js
 - Messages.js
 - Input.js
 - Logout.js
 - Spinner.js
 - UI Folder
 - Button.js
 - ChatUi.js
 - Form.js
 - InputUi.js
 - MessageUi.js
 - ModalStyled.js
- ❖ Hooks Folder
 - useAuth.js
 - useMessage.js
 - useOnline.js
 - useUsers.js
 - useValidation.js
- ❖ Firebase Folder
 - Configuration
 - Context Provider
- ❖ Styles Folder
 - Globals.css
- ❖ Validations Folder
 - validateLogin.js
 - validateSignUp.js

Page - Folder: Carpeta destinada para la rutas en el framework. Contiene las páginas principales de la app, Index.js que será la principal, mostrando los componentes Login y Chat en dependencia de si hay un usuario registrado o no, y Signup.js que es la página para registro de los usuarios en el sistema.

- **Layout.js:** Master Page, contiene toda la información que es reutilizada en la app como por ejemplo un header. También es el lugar perfecto para insertar toda la información seo que se declara dentro de la etiqueta <head> en html.
- **Login.js:** Componente que contiene el formulario para iniciar sesión en la app. Utiliza hooks personalizados para la validación.
- **Chat.js:** Componente principal para la funcionabilidad del Chat, muestra en el lateral izquierdo todos los usuarios registrados en la app, mostrando un indicador rojo/verde que representará si hay algún usuario activo o no. Luego de presionar algún usuario comienza el hilo entre estos o continua el mismo anterior creado, cargando y continuando la conversación anterior. Este componente utiliza hooks personalizados para obtener los usuarios registrados y el estado de los mismos.
- **Messages.js:** Este componente se mostrará si es creado o cargado algún hilo. Utiliza el hook useMessages personalizado para cargar o empezar una conversación, luego de escoger el usuario con el cual se desea chatear. Cambiará el estado del usuario a desconectado en caso de cerrar el navegador desde aquí.
- **Message.js:** Componente para el estilo de los mensajes. Tiene dos estilos (Azul y Gris) los cuales serán representados en dependencia de quien escribe en el chat.
- **Input.js:** Será el input que utilizará un usuario para escribir en el chat.
- **Logout.js:** Botón para cerrar sesión en la app. Cambiará también el estado del usuario a desconectado.
- **Spinner.js:** Será un Styled component spinner que se mostrará cuando se espere por una respuesta del servidor.

Hooks: Carpeta que contendrá todos los hooks personalizados utilizados en la app.

- **useAuth.js:** Manejará y devolverá el usuario actual registrado en el sistema.
- **useMessages.js:** Cargará los mensajes dado el campo por el cual se desea ordenar, y un hilo.
- **useOnline.js:** Será el marque el usuario como desconectado cuando este se desconecte por cualquier motivo.
- **useUser.js:** Cargará y devolverá los usuarios registrados en el sistema.

- **useValidations.js:** Será el encargado de las validaciones en los formularios, además de manejar los submits y los onChange.

UI - Folder: Carpeta que contendrá todos los Styled Components de la app, Botones, Formularios, Inputs, Contenedores div, párrafos.

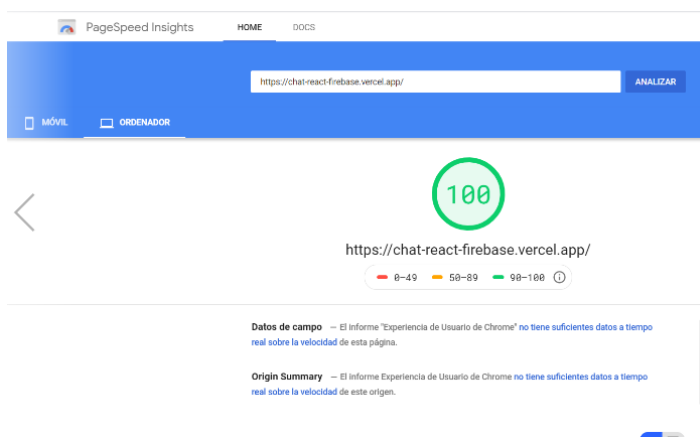
Firebase Folder: Carpeta destinada para la configuración y conexión con el backend Firebase.

Styles Folder: Carpeta propiciada por el framework para declarar los css, en este caso se almacena el global.css que contiene los estilos globales utilizados en la app.

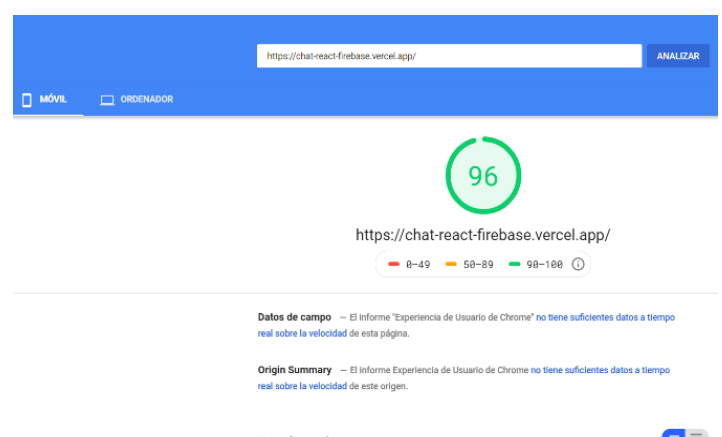
Validations Folder: Carpeta que almacena las validaciones requeridas en los formularios de iniciar sesión y registro en la app.

- **validateLogin.js:** Valida que los campos email y passwords sean correctos en el inicio de sesión de la app.
- **validateSignup.js:** Valida que todos los campos sean correctos en el formulario de registro.

Pruebas: Google Speed Insights



Ordenador



Móvil

Algunos Tipos de Arquitectura de aplicaciones de mensajería

- Arquitectura impulsada por eventos + Simplicidad + Iteratibilidad
- Arquitecturas de terceros como Firebase y Layer
- Arquitecturas frontend para crear una aplicación de mensajería.
- Arquitecturas de chat descentralizadas
- Arquitecturas basadas en plataformas

Arquitectura impulsada por eventos + Simplicidad + Iteratibilidad

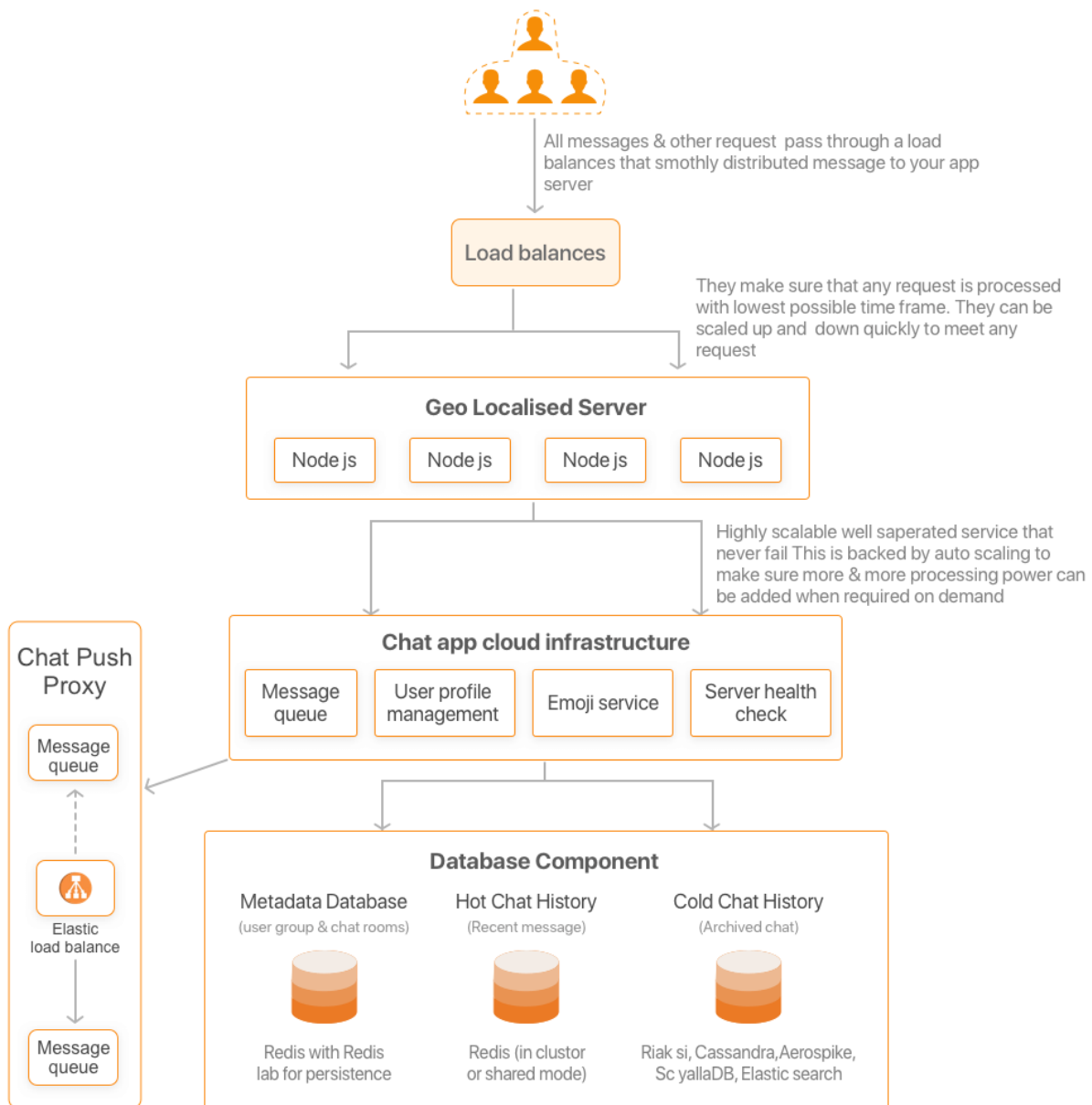
Desafíos comerciales más comunes que enfrentamos cuando desarrollamos un MVP para aplicaciones de chat.

Desafíos que deberían dar forma al backend de un aplicación de mensajería

- **Crecimiento futuro:** evolución de la aplicación
- Capaz de manejar **10k usuarios concurrentes**
- Comunicación en **tiempo real**
- Construirlo en **3-6 meses**
- **Las salas de chat** deben tener el tamaño y la capacidad suficientes
- **Buscar en varios canales** simultáneamente
- **Escala** cuando la carga aumenta repentinamente
- Recuperación más rápida del **historial de mensajes**
- Poder compartir mensajes con **soporte multimedia**

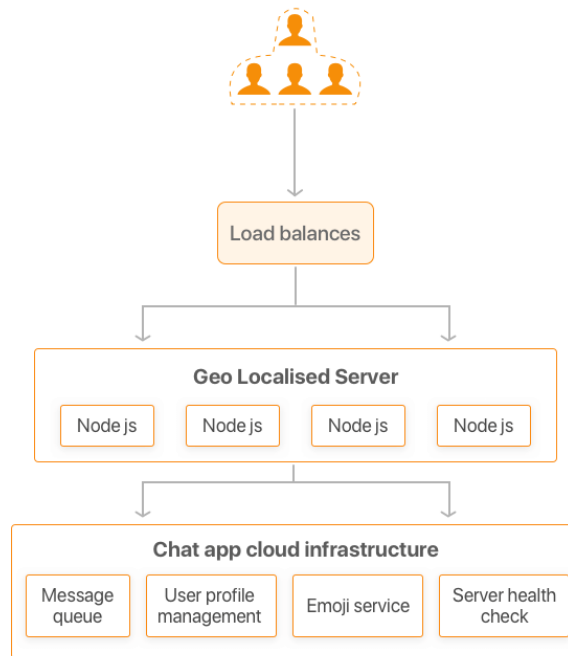
Así es como debería verse su base arquitectónica

Chat Architecture



NodeJS, AWS MQ y AutoScaling

Chat Architecture



Esta es la salsa mágica de la aplicación

Se recomienda usar NodeJS aquí porque, de inmediato, puede manejar 100,000 usuarios concurrentes en un solo núcleo de su servidor. La mayoría de nuestros servidores tienen 5-7 núcleos en estos días. Incluso con optimizaciones más simples, debería poder manejar 1 millón de usuarios con un solo servidor.

Superar el límite de # de usuarios por sala de chat

Las salas de chat tienen un problema, uno que necesita que alguien lo resuelva. Si observa Telegram, Whatsapp, Skype, Slack, etc., encontrará que no puede agregar usuarios más de un límite especificado.

Esta solución puede **alojar alrededor de 70.000 usuarios en una sola sala de chat** sin degradar el rendimiento. También **puede manejar fácilmente 80-200 mensajes por segundo**.

¿Qué es tan especial que hace que este tipo de aplicación sea extremadamente eficiente y escalable bajo cargas pesadas?

- **Disponibilidad regional:** utiliza los servicios Multi AZ de Amazon para obtener la mejor disponibilidad regional
- **Colas de mensajes:** utiliza AWS MQ para asegurarse de que toda la información transferida de las aplicaciones al servidor se ponga en cola, se paralelice y se almacene de manera confiable sin ningún cuello de botella en el rendimiento
- **Escalamiento Auto** - carga adicional del balanceador y reglas AutoScaling en AWS para asegurarse de que si más potencia de cálculo es necesidad AWS automáticamente añada más servidores sin mí necesidad de aprobarlos
- **Separaron las funcionalidades críticas del negocio** y las funcionalidades de terceros cuando escribí la aplicación basada en NodeJS.
- **Serialización JSON:** esto puede parecer pequeño o técnico, pero optimizado y la reducción del tamaño de la información que está comunicando al servidor también reduce una gran cantidad de carga
- Con las aplicaciones de chat, **las notificaciones automáticas son un verdadero problema** si no se hacen bien, la aplicación podría entregar 200 notificaciones automáticas en un día, y separarla con su propia infraestructura le ayudará mucho.

Arquitectura de base de datos para la escalabilidad y el rendimiento de la aplicación

Por lo tanto, antes se muestra sobre la recuperación más rápida del historial de mensajes y las salas de chat. Pero, **si vamos a una hoja de Excel de 500 filas** (no ordenadas alfabéticamente) y buscamos 3 nombres. ¿Qué tan rápido pudiera hacerse?

Actualizar y almacenar información en su base de datos (DB) tampoco es mágico. Sin embargo, los algoritmos son mucho más rápidos en la recuperación de información, pero **con una aplicación de mensajería, las bases de datos rápidamente se vuelven enormes en tamaño y complejidad.** Haciendo muy difícil encontrar, clasificar y actualizar información. Los algoritmos a continuación destacan mejor cómo la recuperación de información puede consumir mucho tiempo cuando elige una forma sobre otra

	 <u>Insertion</u>	 <u>Selection</u>	 <u>Bubble</u>	 <u>Shell</u>	 <u>Merge</u>	 <u>Heap</u>	 <u>Quick</u>	 <u>Quick3</u>
 <u>Random</u>								
 <u>Nearly Sorted</u>								
 <u>Reversed</u>								
 <u>Few Unique</u>								

Para comprender mejor los desafíos del nivel de base de datos, primero echemos un vistazo al tipo de información que normalmente almacenamos:

- **Información de la sala de chat:** nombre de la sala de chat, quién está en esta sala de chat, etc.
- **Mensajes recientes:** estos son mensajes que se acaban de enviar
- **Mensajes archivados:** **mensajes** antiguos por lo general. Pero puede decir formalmente que cualquier mensaje que no esté en los últimos 50 mensajes es un mensaje archivado.

Ahora, también se necesita de una canalización que pueda enviar información de manera confiable a su base de datos.

Entonces, tenemos cuatro preocupaciones diferentes cuando se trata de la parte de la base de datos de la arquitectura de su aplicación.

Consideraciones de ingeniería:

- Un usuario sentiría una mala experiencia de usuario si no puede encontrar los **últimos 50 mensajes** en su aplicación.
- Un usuario querría enviar información tan rápido al servidor. **¡El chat en tiempo real** es la norma ahora!

- **Las salas de chat requerirían su propio almacén de datos**, ya que almacenar esta información en la misma base de datos sería fatal para el rendimiento y la escalabilidad de su aplicación.
- **Se publicarían más de 100.000 a 200.000 mensajes a diario** si su aplicación tuviera más de 100.000 usuarios.

Es bastante común que las aplicaciones de chat adquieran usuarios rápidamente y escalen. GoChat y GoSnap consiguieron sus primeros 100.000 usuarios en tan solo unos días, ¡de forma orgánica!

La configuración de la base de datos para la escalabilidad y el rendimiento del chat incluiría:

- Base de datos de metadatos
- Base de datos de historial de chat caliente
- Base de datos del historial de Cold Chat
- Cola de mensajes frente al historial de chat frío

Para la base de datos de metadatos, se puede usar cualquier almacén de valor clave que tenga una estructura de datos rica. Incluso puede usar Redis si se usa con RedisLabs para la persistencia.

Para la base de datos de Hot Chat, Redis se puede usar en un clúster o en modo compartido. También puede considerar Aerospike, ScyllaDB o Elasticsearch.

Para el almacenamiento de datos de Cold Chat, Riak es una de las opciones más populares. Pero también puede usar Cassandra / ScyllaDB, AeroSpike.

¿Qué rendimiento se puede esperar de esta configuración?

Se puede asegurar que pueden esperar este trabajo sin problemas con:

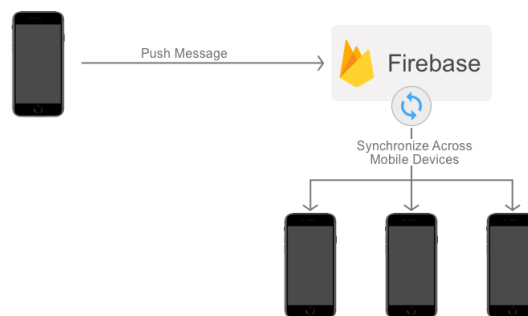
- 15 millones de mensajes / día
- 200 mensajes / seg

Lo suficientemente bueno para apoyarse y lo suficientemente simple como para que no cueste la tierra construirlo.

Arquitecturas de chat basadas en plantillas: Firebase y Layer

Las aplicaciones de chat creadas sobre Firebase son un gran ejemplo de este tipo de arquitecturas. Podemos ver a Firebase como un conjunto de reglas preconfiguradas para construir las aplicaciones donde solo tiene que trabajar en el frontend, la mayor parte de la lógica del backend de la aplicación ya estaría construida. Los problemas de escalabilidad anteriormente destacados, con Firebase, no será necesario resolverlos, pues Firebase se encargará de la mayoría de las cosas por uno.

Así es como se ve una arquitectura de aplicación de mensajería que usa Firebase



Eso es todo. Sencillo, eficiente y rápido

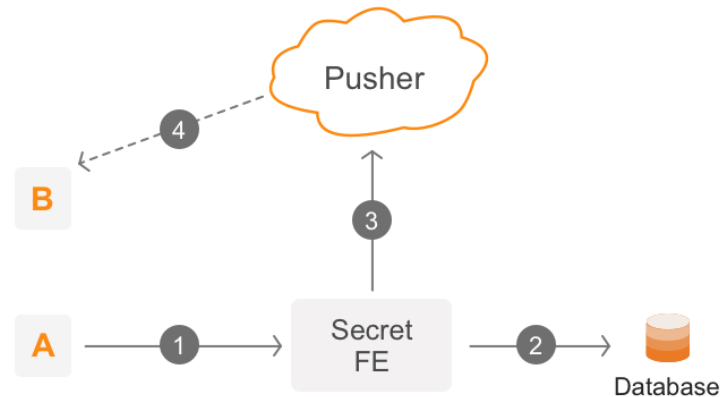
Otra plataforma que trae una base prefabricada a la mesa, es Layer.io. Son similares a Firebase, pero también proporcionan elementos de IU prediseñados.

Hubo un momento en el que los creadores de Secret planeaban crear una solución de chat en tiempo real. Comenzaron con preocupaciones más o menos similares a las que he enumerado antes. Se quedaron con tres opciones:

- Utilice un servicio de chat de terceros como Layer
- Utilice una implementación de websocket de terceros como Pusher
- Cree su propio protocolo basado en websocket sobre AWS

El equipo de Secret rápidamente terminó **rechazando** a Layer como una posibilidad, ya que no pudieron **iterar o personalizar lo suficiente**. Además, también querían un control total sobre sus datos. Secret decidió ir con Pusher para construir un sistema

basado en websocket altamente escalable. Así es como se veía su arquitectura de alto nivel.



Esta solución se amplió fácilmente a más de 1.000.000 de conexiones simultáneas. Fue rápido, rápido y escalable.

Cuando optar por una solución tecnológica como Pusher, Firebase o Layer:

- Si su equipo es un negocio pesado que espera una sobrecarga de tecnología muy baja, estas arquitecturas de terceros son la mejor opción para usted.
- Si usted es un equipo de tecnología que solo quiere lanzar un prototipo más rápido, esta arquitectura de terceros es algo que podría considerar aquí.

Algunas precauciones que deben tomarse al evaluar estas plataformas de terceros:

- En la mayoría de los casos, controlan tus datos.
- El sistema suele ser rígido e inflexible, lo que dificulta la adopción de nuevos cambios personalizados.
- A veces, terminaría pirateando todo el sistema solo para asegurarse de que puede obtener incluso una pequeña función personalizada
- Aparte de lo que ya está ahí, el sistema no le permitirá superar los puntos de referencia en términos de rendimiento.

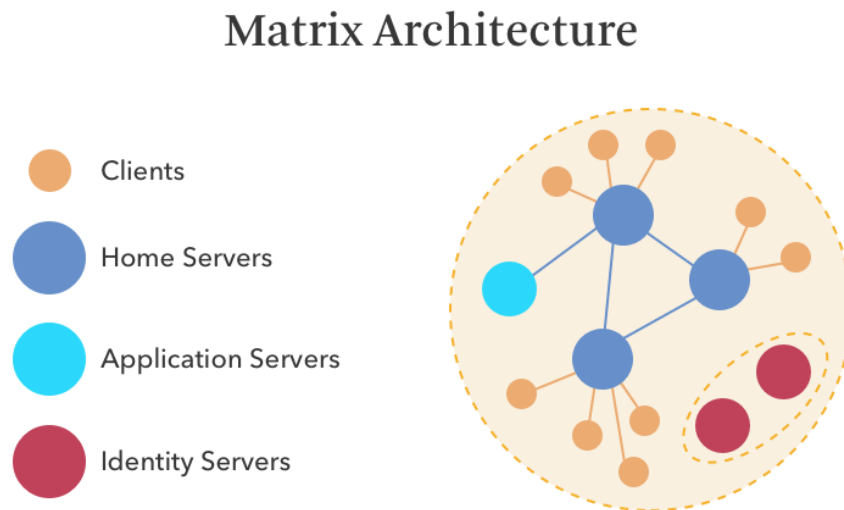
Arquitecturas de chat descentralizadas

Hasta ahora, las antes expuestas son algo llamado arquitecturas de chat centralizadas. Las arquitecturas descentralizadas se basan principalmente en la parte superior de Blockchains o Distributed Ledgers para admitir servicios de chat seguros descentralizados.

Utilicemos Matrix.org para crear una aplicación de chat B2B en redes descentralizadas. Matrix como solución se centrarían en:

- Grupo de chat
- Señalización WebRTC
- Reducción de silos

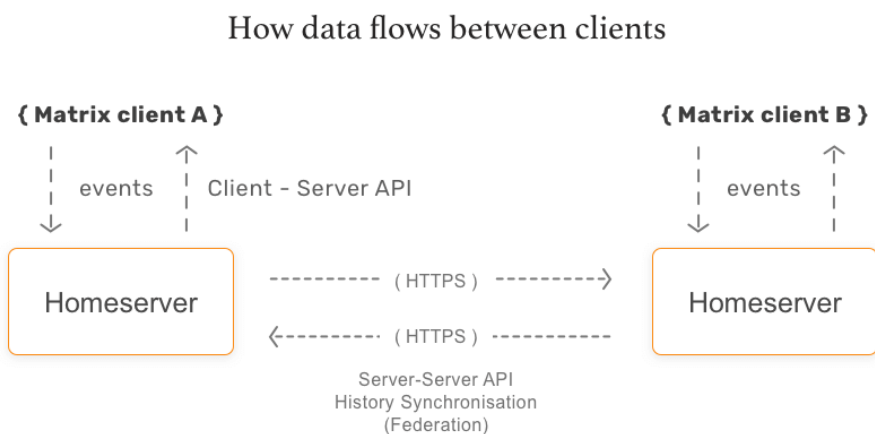
En lugar de estar alojado en un servidor centralizado, así es como se ve realmente la arquitectura basada en Matrix



Esta arquitectura en general es una malla de servicios. Lo realmente interesante de esta arquitectura es que el historial de conversaciones se distribuye entre estos servidores. No hay un único servidor que sea dueño de la conversación.

Creo que un gran problema que existe en la mensajería es la falta de protocolos y estándares abiertos que puedan facilitar esta comunicación. Matrix simplifica esta comunicación utilizando WebRTC para comunicar información.

Si bien la mayor parte de la arquitectura general de una aplicación que se crea aquí es similar a la que tenemos en las soluciones de chat centralizadas, la diferencia radica en cómo los nodos de chat descentralizados interactúan entre sí. Eche un vistazo al siguiente ejemplo que muestra dos nodos descentralizados (servidores domésticos) hablando entre sí.



Si esto parece demasiado confuso. Observemos algunos de los conjuntos prácticos para crear una aplicación de matriz:

- Comprar un nombre de dominio: como Slack.com
- Obtener un alojamiento (su propio servidor): esto iniciaría los servicios
- Con las herramientas de línea de comandos de Linux, instalar Synapse en el servidor
- Ahora se tiene lo que se le llama un servidor doméstico
- Escribir la otra parte de la lógica de la aplicación: esto incluiría todo, desde cómo se vería su aplicación hasta cómo va a aceptar usuarios en una sala de chat

Una parte más difícil de crear aplicaciones de chat descentralizadas son las pruebas. Casi todas y cada una de las aplicaciones de chat descentralizadas son únicas, y eso hace que probarlas sea extremadamente difícil. La prueba de carga de una aplicación descentralizada que se creó desde cero para resistir el spam en la red nuevamente es una nuez difícil de romper.

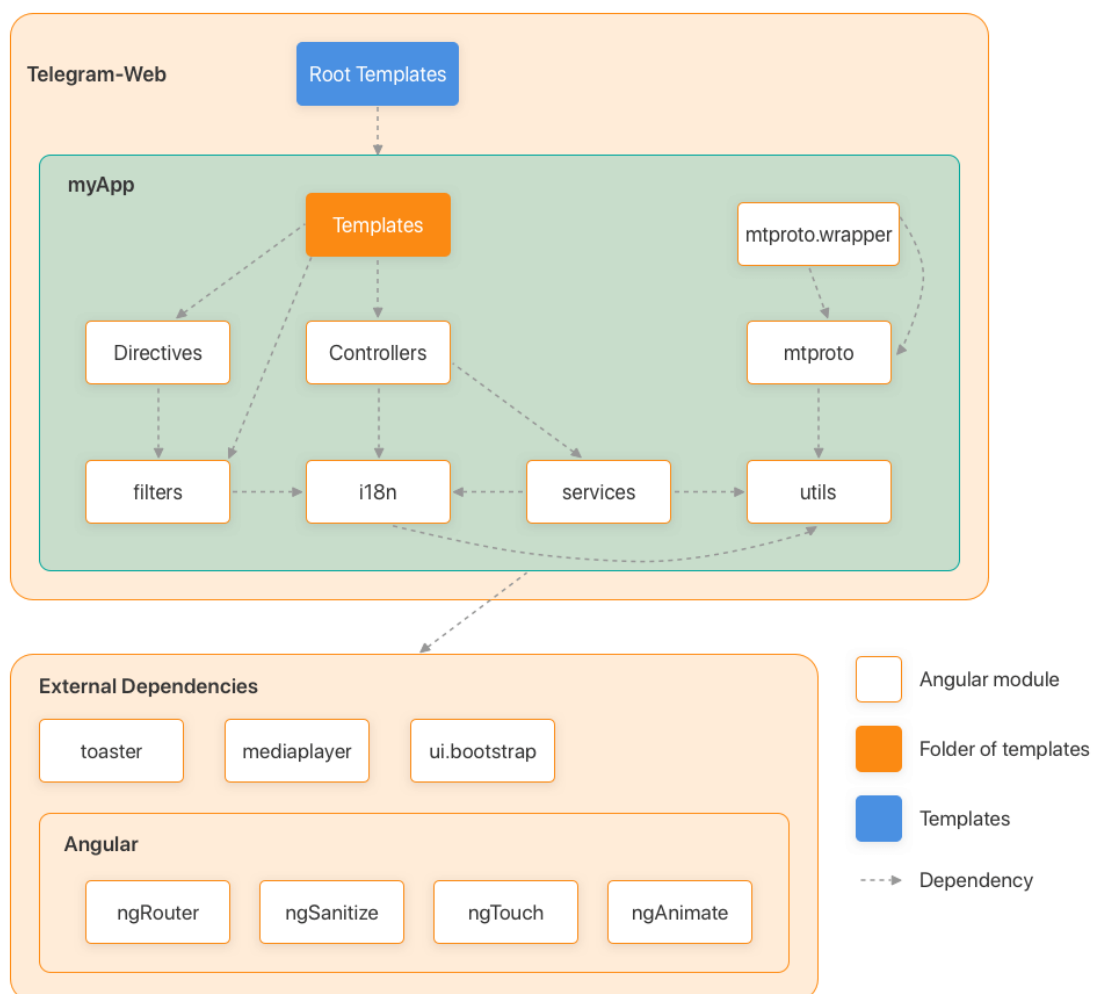
Arquitecturas frontend para crear una aplicación de mensajería

Dos de los marcos tecnológicos principales que los programadores usan para construir el frontend del chat son:

- Angular
- React

Arquitectura frontend de la aplicación de chat basada en Angular

Un diagrama arquitectónico de alto nivel del cliente para una aplicación de mensajería en Angular se vería así:



Como se puede observar, hay reglas internas, externas y de capa realizadas para construir una interfaz web de alto rendimiento.

Una nota importante: Cuando se habla de arquitecturas, hablamos de separar diferentes servicios en diferentes preocupaciones. Telegram en realidad comenzó como un proyecto de pasatiempo, la falta de estas separaciones rápidamente hizo que fuera difícil mantener y agregar nuevas funciones. En el mundo de los desarrolladores, podemos definir esto como una deuda arquitectónica.

El frontend propuesto basado en Angular sufriría una deuda arquitectónica similar si las cosas no están bien separadas en su API.

Deuda técnica con aplicaciones de chat basadas en Angular

Cuando las cosas no se hacen correctamente, especialmente con proyectos complejos de frontend, las cosas pasan rápidamente de manejables a un infierno de administración.

A medida que estos proyectos avanzan y uno se enfoca en la retención y adopción de usuarios, notará muchas cosas como muchas "Tareas pendientes" y "Arreglar después" en el sistema de gestión de proyectos.

Si bien estos se hicieron de buena fe, dado el ritmo de funcionamiento general de la ingeniería, lo que queda a menudo se deja.

Esto hace que el producto sea cada vez menos estable y menos escalable con un rendimiento que solo se puede soñar sin una reescritura importante. Pensar en esto como una hipoteca con tasas de interés que casi se duplican en el momento en que no realiza ni un solo pago.

Cómo reducir la deuda técnica

Algunos principios generales (pero efectivos) que reducen fácilmente la deuda técnica son:

- Mejorar la cobertura de las pruebas

- Dividir archivos de código grandes en archivos pequeños dedicados
- Seguir migrando a la siguiente versión más estable
- Agregar proceso de integración continua

Deuda de documentación inicial

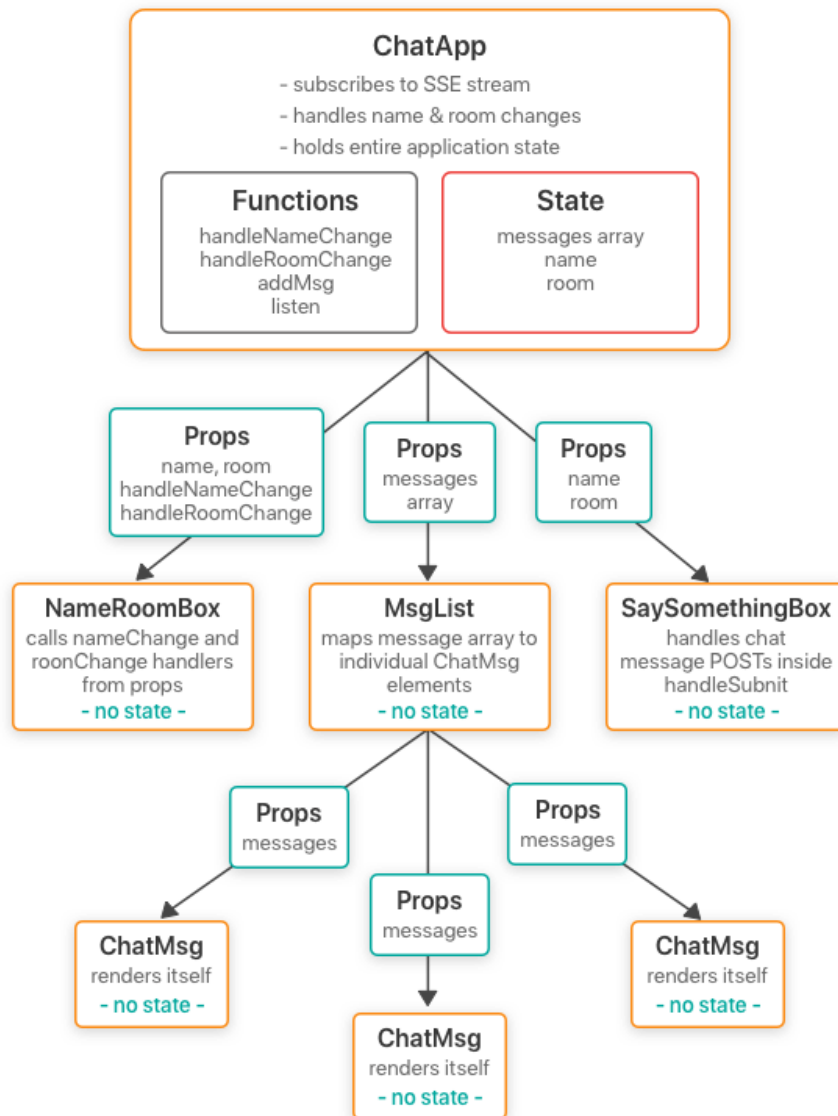
La deuda técnica no ocurre de forma aislada, más a menudo se desencadena por la falta de documentación inicial. Es por eso que es recomendable comenzar proyectos complejos como desarrollar una aplicación de chat con una documentación actualizada.

Arquitectura frontend de la aplicación de chat basada en React

React funciona fundamentalmente de diferentes maneras que la mayoría de los otros marcos basados en web.

Esta arquitectura a diferencia de la angular tiene muchos beneficios:

- Relativamente más rápido.
- Puede alcanzar 60FPS sin sudar incluso en el móvil. 60FPS es lo que hace que su usuario sienta que la aplicación funciona. Cualquier cosa a continuación que haga que la aplicación parezca lenta en rendimiento.
- Conceptualmente muy simple.



Pero si fuera así de fácil, todos habríamos estado creando nuestras aplicaciones de chat en React. Pero, todos divagamos cuando se trata de la selección de tecnología y marco.

Si hablo específicamente de una aplicación de chat realmente grande y compleja, consideraría React dado que:

- Hay toneladas de mejores prácticas para la organización de grandes aplicaciones.
- Gran soporte para pruebas automatizadas y generación de compilación
- Nuevamente, depende de su proyecto y sus necesidades únicas.

Cómo hacer una aplicación de mensajería: características y selección de plataforma

Antes de pasar al negocio real de la creación de aplicaciones, se debe tener una idea clara de lo que desea crear. La selección de plataformas juega un papel importante en el proceso de creación de aplicaciones, puede romper o hacer que su aplicación. La selección incorrecta de la plataforma le costó mucho en términos de recursos y tiempo a largo plazo. Entonces, antes de ensuciarnos las manos, echemos un vistazo a la viabilidad y la implementación de cada función en diferentes plataformas.

Videollamada

Si bien la videollamada es algo que se está volviendo extremadamente común en estos días con la mayoría de las aplicaciones de mensajería, la implementación de la función de chat de video en mi opinión requiere mucha consideración.

En primer lugar, la forma en que funciona React native, considere lo siguiente:

- Si su aplicación solo va a ejecutar la funcionalidad de videollamadas en un momento específico, entonces puede considerar usar React native como tecnología subyacente para su aplicación.
- Pero si tiene una notificación, emojis en vivo o cualquier otra forma de interacción, React native no podrá entregarse correctamente.

¿Por qué?

React native usa algo llamado "puente", esto es lo que ayuda al código de su aplicación React nativa a comunicarse con los componentes nativos del móvil. Y una vez que se inicie esta comunicación, este puente no permitirá nada más. Por ejemplo, si este puente se utiliza para videollamadas, no permitirá que pasen ni siquiera los emoji. Ahora, cuándo si o cuando no se debe seleccionar React Native.



Si bien las videollamadas ya eran difíciles, lo que la mayoría de las empresas emergentes y las empresas ignoran al principio es la idea completa de la interfaz de usuario. Esta interfaz de usuario no es solo una función aleatoria, esta interfaz de usuario se trata de hacer algo interactivo, poder leer y hacer referencia a mensajes y mucho más.

Imagínese si un equipo de UX dice que necesita hacer que su aplicación sea más interactiva, pero lamentablemente eligió la tecnología incorrecta al principio. ¿Qué se pudiera hacer entonces? ¿Desechar todo lo que has creado en React native solo para que tu aplicación sea más interactiva?

Es por eso que para el video definitivamente debería comenzar con tecnologías nativas. Incluso si elige React native, puede fusionar una contraparte nativa donde el código nativo de React no funcionará.

Sincronización automática de contactos

La sincronización de contactos es un proceso continuo y requiere una conexión persistente con el servidor.

Siempre que haga clic en cualquiera de las aplicaciones de chat, debe haber encontrado un cuadro de diálogo que muestra **Sincronizando**. Básicamente, está sincronizando sus datos locales con los datos del lado del servidor.

Este proceso también puede ejecutarse en segundo plano sin su conocimiento (pero con su consentimiento), lo que puede consumir mucho la batería y requerir una gran cantidad de potencia de procesamiento.

La sincronización de contactos es una característica básica y se trata en profundidad tanto en nativo como en React native. Puede elegir cualquiera de las plataformas según sus requisitos.

Crear la funcionalidad de sincronización de contactos es fácil, ya sea nativo o React native. Hay toneladas de ejemplos disponibles, y esto no es fácil de implementar, pero también es fácil de implementar con rendimiento.

Indicador de escritura en una aplicación de chat

Para implementar un indicador de escritura en su aplicación de chat, generalmente recomiendo el patrón de mensajes suscriptor - editor. Aquí, un receptor se suscribirá al estado del remitente y el editor publicará el estado del remitente.

Tenga en cuenta que estos indicadores de escritura son pesados en el servidor de su aplicación. He visto casos en los que han ocupado hasta el 95% de todo el tráfico de mensajería, cuando los mensajes reales solo representaban el 5%. Al implementarlos, tenga en cuenta que debe optimizarlos con mucho cuidado o que alguien se encargue de ello por uno.

Cuando se habla de crear una interfaz de usuario de indicadores de mecanografía, los indicadores de mecanografía basados en aplicaciones nativas superan significativamente a las contrapartes que hemos construido sobre React native. La razón principal por la que los indicadores de tipeo funcionan mal en las aplicaciones de chat es porque la representación de esta interfaz de usuario en React nativo causa problemas.

Su implementación nativa, como en Android o iOS, tiene una funcionalidad incorporada y los oyentes de reacción también proporcionan su implementación en react nativo. Pero ir con SDK de terceros (Pusher, Jiver, Applozic, etc.) en forma nativa le brinda una mejora del rendimiento y la capacidad de escalarlo mejor en el futuro.

Las plataformas nativas tienen algunas funcionalidades que facilitan aún más la creación de estos indicadores de escritura para aplicaciones de chat. Pero, si es serio y tiene la intención de crear una aplicación para escalar, también debe considerar soluciones de terceros como Pusher.

Uso compartido de archivos grandes

En la aplicación de chat, necesita comunicación en tiempo real, es decir, no dependería únicamente del cliente para decidir cuándo obtener datos del servidor. En su lugar, el servidor enviará los datos al cliente cuando estén disponibles.

Esto se puede lograr fácilmente usando socket.io.

La buena noticia es que está disponible para la plataforma nativa y react native.

Si sus únicas consideraciones son:

- Tamaño de archivo para compartir.
- Velocidad de carga y descarga.

Está bien para ir con cualquier native o React Native, pero en caso de que sea fácil de implementar, React Native supondrá un gran desafío. Bueno, a nadie le gustaría hacer ese trabajo pesado y consumir recursos innecesarios. Entonces, Native es la apuesta para ir.

En la cámara de la aplicación

La integración de la cámara es una de las tareas más complicadas en el desarrollo de aplicaciones. Tienes que cuidar su capacidad de respuesta, número de fotogramas / seg en un video y calidad de imagen capturada con sus filtros.

Función de historias de tipo Instagram

Las historias son básicamente animaciones. Hay muchos tutoriales disponibles para mostrar cómo crear un cubo 3D utilizando el poder de CSS.

"StoriesProgressView" nos brinda la capacidad de implementar estados como WhatsApp o historias como Instagram en su aplicación.

Al usar MobX con react native, puede implementar la función de estado al igual que WhatsApp.

Encriptado de fin a fin

Las características de seguridad han abandonado una implementación similar tanto en react como en nativa en términos de recursos y complejidad. De acuerdo con su requerimiento como disponibilidad de desarrolladores y nivel de implementación, puede elegir cualquiera de ellos.

La API de Signal se puede utilizar para implementar el cifrado de extremo a extremo en la plataforma nativa.

La biblioteca incorporada "crypto-js" proporciona cifrado de extremo a extremo en React native. Signal API también está disponible con React Native para la implementación de cifrado de extremo a extremo.

Notificación de inserción

Las notificaciones push son la función más utilizada en la aplicación de chat. Mantendrá a sus usuarios comprometidos con la aplicación. Imagine que ha trabajado lo suficiente en su aplicación, la lanzó al mercado y vio crecer su base de usuarios.

Pero al día siguiente te despiertas y notas que la participación de los usuarios se redujo a unos pocos miles de millones. La aplicación funciona bien, pero el problema con las notificaciones puede causar un gran desastre, más aún si su aplicación está en MVP.

Los problemas más común que enfrentamos con las notificaciones automáticas son:

1. La notificación push se detiene cuando la aplicación está en segundo plano.
2. No recibirá notificaciones de las duraciones cuando estuvo desconectado.
3. Puede interferir con la interfaz de usuario de su teléfono.
4. A veces, los usuarios dejan de recibirlos cuando actualizan su sistema operativo.

Estos problemas pueden reducir drásticamente su base de usuarios.

Implementarlos con React Native puede ser un desafío considerando su naturaleza de un solo hilo. Su rendimiento también puede diferir con respecto a Android e iOS.

En caso de ser nativo, se puede implementar mediante Google Cloud Messaging (GCM) o Firebase Cloud Messaging (FCM).

No es muy recomendado con GCM, aunque es de Google. Las aplicaciones populares como WhatsApp y Facebook tienen sus propios protocolos para implementar. Si tiene suficientes recursos y un presupuesto decente para hacerlo, muy bien, hágalo. De lo contrario, puede considerar usar FCM o Expo.

Actualizaciones de la pantalla de chat en tiempo real

WebSockets, tanto en Android como en iOS, proporcionan conexión en tiempo real al servidor y actualizan la pantalla de chat en tiempo real. Los WebSockets también están disponibles para que React implemente la función.

La actualización del chat en tiempo real puede plantear algunos desafíos a la interfaz de usuario con la reacción, mientras que en el caso de la aplicación nativa es fácil y sin problemas.

Arquitecturas basadas en plataformas estándar

Pasamos por el proceso paso a paso para crear una aplicación de mensajería perfectamente escalable.

Existen diferentes proveedores de SDK que brindan integración de extremo a extremo para implementar tantas funcionalidades como uno pueda pedir. Pero depende de la elección, considerando los requisitos de escalabilidad, rendimiento, número de usuarios a los que desea atender y presupuesto del proyecto.

En la etapa inicial, no se recomienda saltar directamente para luchar contra los grandes, como Whatsapp, Line, Viber, etc., en el mercado. Sería aconsejable empezar despacio y construir sobre la marcha.

Para comparar diferentes soluciones, veremos su costo de licencia, costo de alojamiento, costo de instalación e integración y, finalmente, costo de personalización.

Consideraríamos estos proveedores en tres escenarios:

1. Una aplicación que se está iniciando.
2. Una aplicación que tiene 10K usuarios diarios.
3. Una aplicación que tiene 100.000 usuarios diarios.

Estos son algunos de los principales proveedores de SDK que vale la pena considerar:

Quickblox

	Started Free	Advanced \$49pm	Pro \$219pm	Small \$599pm	Medium \$1,199pm	Large \$2,399pm
Hosting	Shared Cloud			Dedicated Instance(Enterprise)		
Chat messages	20/s	35/s	50/s	200/s	1000/s	∞
Push notification	20/s	35/s	50/s	200/s	1000/s	∞
Monthly active users	20,000	35,000	75,000	250,000	500,000	∞
Support channels	Community	Community	+Tickets +Email	+Tickets +Email +Phone	+Tickets +Email +Phone +Acc. manager	+Tickets +Email +Phone +Acc. manager
Support hours	—	—	9am-5pm	9am-5pm	24/7	24/7
Integration hours				5hrs	10hrs	20hrs







No cobran ninguna tarifa de licencia.

La desventaja de Quickblox SDK es que es muy básico, por lo que debe escribir una interfaz de mensajería completa, lo que puede demorar entre 2 y 3 meses.

PubNub

	FREE	PILOT	LAUNCH	SCALE	PRO
	\$0/mo	\$149/mo	\$399/mo	\$799/mo	Custom Pricing
INCLUDED USAGE					
Daily Active Devices ?	100	1000	5,000	20,000	100 to Millions
Message Included ?	1 Million	5 Million	10 Million	40 Million	1 Millions to Billion
Additional Messages ?	Available with Pilot	\$2.50 / Million	\$2.50 / Million	\$2.50 / Million	Customizable / Million
SERVICE					
Publish/Susbscribe ?	✓	✓	✓	✓	✓
Access Manager ?	✓	✓	✓	✓	✓
Additional Add-ons ?	Free	\$37/mo	\$99/mo	\$199/mo	Custom per Service
PubNub BLOCKS ?	BLOCKS is free until Dec. 1 (up to 500M Executions). Afterward BLOCKS pricing is \$9/million Event Handler executions.				

Aribista

Sandbox	Startup	Pro	Business	Premium	Enterprise
					
Free	\$49 per month	\$99 per month	\$299 per month	\$499 per month	A Tailored Soution
100 Max Connections ?	500 Max Connections	2000 Max Connections	5000 Max Connections	10000 Max Connections	10000+ Max Connections
Unlimited Channels	Unlimited Channels	Unlimited Channels	Unlimited Channels	Unlimited Channels	Unlimited Channels
200k Messages/Day ?	1 million Messages/Day	4 million Messages/Day	10 million Messages /Day	20 million Messages /Day	20 million Messages /Day
Limited Support ?	Limited Support	Limited Support	Standard Support ?	Standard Support	Premium Support ?
SSL Protection	SSL Protection	SSL Protection	SSL Protection	SSL Protection	SSL Protection

Mientras que los proveedores anteriores proporcionan backend de mensajería, Pusher ofrece comunicación de mensajes en tiempo real. Le proporciona un canal para enviar datos entre dos usuarios.

Pusher cobra tanto por conexiones activas como por la cantidad de mensajes por día. Suponiendo que un usuario pasa 10 minutos enviando mensajes por día, eso significa que cada conexión simultánea puede admitir 150 usuarios diarios. Esto significa que el plan gratuito podría admitir 15.000 usuarios diarios.

Pero hay una advertencia aquí. Dado que Pusher es un servicio de entrega de datos, debe crear su propia interfaz de mensajes y el protocolo de mensajes completo. Esto puede tardar entre 2 y 3 meses.