Universidad Zaragoza

Trabajo Fin de Grado

Librería de mensajería instantánea fiable para aplicaciones móviles (Chat-Sot)

Autor:

Razvan Gabriel Oniga Rus

Director:

Iván Verde Pita

Grado Ingeniería Informática
Escuela Universitaria Politécnica de Teruel.

Noviembre 2021

Resumen

Con el avance tecnológico, surgió la mensajería instantánea, inicialmente implantada en ordenadores, con aplicaciones como Messenger. Posteriormente, como la gente percibió que esta modalidad de comunicarse era muy cómoda, rápida y barata, lo que hizo que se extendiera también a los dispositivos móviles con aplicaciones como WhatsApp y muchas otras.

Así pues, a lo largo de los años, los chats han ido ganando presencia en todo tipo de ámbitos: servicios de compraventa, consultorías, atención al cliente, redes sociales o simplemente un servicio de mensajería.

En este contexto se ha creado una librería, que permite a cualquier equipo de trabajo, integrar un chat en su cliente y servidor en apenas unas horas. La finalidad de la librería es reutilizar soluciones ya implementadas para ahorrar el coste del desarrollo. Dicha librería dispone de chat individual, chat grupal, control de mensajes leídos y un estado de disponibilidad del usuario.

Respecto a los chats, se tratan de conversaciones donde se intercambian mensajes de texto entre dos usuarios en caso del chat individual y entre varios usuarios en un chat grupal.

En los grupos, no está delimitado el número de miembros que puede abarcar, esto se deja a disposición de la solución final, de igual forma que la política que se le aplica a la creación y administración de grupos, es decir, si se les permite a los usuarios crear grupos, o los grupos son gestionados por usuarios administradores.

El control de mensajes leídos incluido en esta librería es a modo informativo, enfocado al usuario; donde si el check es negro, se indica al usuario que no ha leído la conversación, y por el contrario, si es azul, sí. Por otro lado, también se dispone de un sistema de estados personales, lo que indicará a los demás si estas disponible u ocupado.

Con estas características se plantea una herramienta rápidamente integrable en la solución final y que una vez operativa, tiene gran margen de personalización.

Abstract

With technological advancement, instant messaging emerged, initially implemented on computers, with applications such as Messenger. Later, as people perceived that this way of communicating was very comfortable, fast and cheap, it was also extended to mobile devices with applications such as WhatsApp and many others.

Thus, over the years, chats have been gaining presence in all kinds of areas: purchase and sale services, consultancies, customer service, social networks or simply a messaging service.

In this context, a library has been created that allows any work team to integrate a chat on their client and server in just a few hours. The purpose of the library is to reuse solutions already implemented to save development costs. This library has individual chat, group chat, control of read messages and a user availability status.

Regarding chats, they are conversations where text messages are exchanged between two users in the case of individual chat and between several users in a group chat.

In groups, the number of members that can be included is not delimited, this is left at the disposal of the final solution, in the same way as the policy that is applied to the creation and administration of groups, that is, if they are allowed Users create groups, or the groups are managed by admin users.

The control of read messages included in this library is informative, focused on the user; where if the check is black, the user is indicated that he has not read the conversation, and on the contrary, if it is blue, yes. On the other hand, there is also a personal status system, which will indicate to others if you are available or busy.

With these characteristics, a tool that can be quickly integrated into the final solution is proposed, and once operational, it has a great margin for customization.

Abreviaturas

API: Aplication Program Interface

BD: Base de Datos

SDK: Software Development Kit

CUL: Casos de Uso Librería

CUU: Casos de Uso Usuario

CUS: Casos de Uso Servidor

DTO: Data Transference Object

AWS: Amazon Web Services

Glosario de Términos

Check: Este término hace referencia al símbolo de confirmación de lectura de una conversación, el cual se utilizará a lo largo del documento para explicar su comportamiento dentro del listado de conversaciones.

Push: A lo largo del documento, se utilizará este término para hacer referencia a las notificaciones cuyo estilo de mostrarse en pantalla es emerger desde la parte superior de la pantalla.

WebSockets: es una tecnología que hace posible abrir una sesión de comunicación interactiva entre el navegador del usuario y un servidor.

Sandbox: hace referencia a un entorno de pruebas de un servicio donde está permitido hacer todo tipo de pruebas para testear funcionalidades que se quieran implementar.

Tabla de contenido

Tabla de Ilustraciones

Tabla de Anexos

Introducción

Comunicarse a través de internet mediante mensajes es más común cada día. Por ello, incorporar un sistema de mensajería en nuevas aplicaciones o servicios es cada vez más demandado.

Por ello, se ha decidido crear esta librería de chat, que proporcionan beneficios en los ámbitos temporales, de coste y reutilización, explicados a continuación:

Tiempo

Cuando se está desarrollando una aplicación se debe tener en cuenta el coste temporal de su implementación y diseño. Entonces, si el servicio de chat es tan habitual, ¿por qué se tiene invertir recursos en desarrollarlo nuevamente para cada ámbito donde se quiera utilizar? ¿Por qué no invertir el tiempo de desarrollo para nuevas funcionalidades o funcionalidades específicas de cada aplicación? La solución perfecta para que los equipos de trabajo no tengan que invertir tiempo en el desarrollo de un chat propio, es el uso de una librería, que permita en unas pocas horas integrar el servicio.

Costes

Relacionado con el anterior objetivo, en el ámbito de un equipo de trabajo, cuanto más tiempo se dedique a una funcionalidad, mayor coste económico tendrá. Si aprovechamos la librería que estamos presentado, obtendríamos un gran ahorro en recursos ya que no solo nos ahorraríamos el coste de un proyecto, sino de todos los que necesiten un chat.

Comodidad y reutilización

Finalmente, los últimos objetivos que se pretende solventar con esta herramienta son la comodidad y la reutilización. Se tiene que evitar reinventar la rueda. Si hay una herramienta que ya desempeña el trabajo que nosotros estamos buscando, hay que aprovecharla. De esta manera, cada vez que se necesite incorporar un chat en alguna aplicación, ya tenemos la solución y, además, de lo único que nos vamos a tener que preocupar es de la personalización, porque se trata de una herramienta universal y con gran libertad para dejarla a tu gusto.

Análisis de las soluciones existentes

En los últimos años, todos los ámbitos han tendido a digitalizarse, lo cual ha llevado a que muchos comercios y servicios cotidianos o no tan cotidianos, se realicen de forma online. Al principio vía web y posteriormente mediante móvil tanto en dispositivos Android, iOS u otras plataformas que no han tenido tanta demanda.

Hoy en día, incorporar un chat en cualquier ámbito es muy común, por lo que constantemente se van a tener que desarrollar estos servicios de mensajería.

Algunos ejemplos de casos donde se utilizan chats:

- A la hora de realizar una consulta o demanda sobre un servicio y/o producto. Por ejemplo, informarse sobre un gimnasio al que se quiere apuntar. Se puede realizar mediante un chat en vez de una llamada, lo que es más cómodo y rápido.

- Al realizar una llamada a una entidad telefónica para llevar a cabo alguna gestión, vamos a tener que lidiar un robot preguntándonos nuestros datos y aguantar una música repetitiva hasta que nos conteste un operador. Con un chat, es todo más sencillo y cómodo para el usuario, si este es capaz de resolver sus demandas en menos tiempo.

- Aplicaciones de compraventa de productos y/o servicios, como Wallapop, Vinted, Uber las cuales también están en auge y todas ellas disponen de chat.

- Una red social, donde comunicarse con tus contactos es imprescindible.

Cada vez que se quiere implementar una aplicación que contenga un chat, es común que el equipo desarrollador, deba destinar recursos a diseñar y programar la misma funcionalidad repetidas veces, pero para diferente ámbito.

Al ser un servicio muy común y demandado, hay muchas opciones que dan solución al problema actualmente en el mercado, pero todas tienen algún inconveniente. Por ello, se van a analizar algunas opciones y posteriormente se van a comparar con el sistema creado para analizar cuál es la diferencia con estas soluciones y las ventajas que ofrece nuestra librería. Se va a tomar en cuenta el método de integración que tienen, las plataformas para las cuales están disponibles, el nivel de personalización y el precio.

PUSHER

Es un servicio alojado en la nube, que ofrece una fácil integración del chat a través de websockets a web y aplicaciones móviles. El funcionamiento resumido de este servicio es que la aplicación móvil realiza una petición al servidor dedicado y este a su vez realiza una petición a los servicios de Pusher que gestiona la petición y devuelve el resultado. Ofrece multitud de opciones para el front-end (Aplicación Usuario) y el back-end (Servidor):

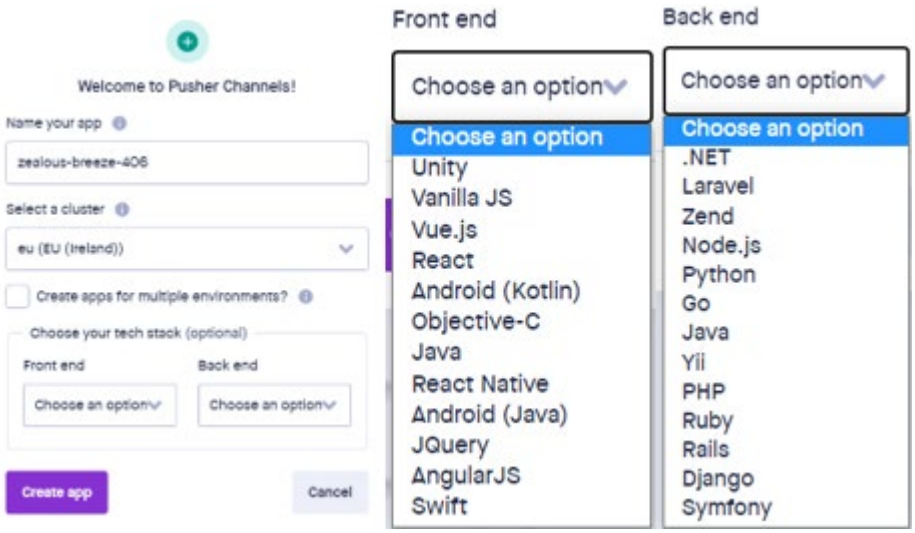


IMAGEN 1 PUSHER: MENÚS DE CREACIÓN CON DISTINTAS OPCIONES

Para implementar este servicio se nos proporciona una guía rápida con la cual se puede probar el funcionamiento del producto, pero para un uso más personalizado se requiere consultar la documentación, la cual es algo minimalista porque no sigue una explicación lineal, sino que tiene apartados para cada elemento aparte. La impresión que da es que, si se necesita un servicio de mensajería básico sin personalización alguna y estar despreocupado, es una buena opción debido a que la puesta en marcha parece costar un tiempo bastante corto. Pero si lo que se está buscando es tener un control mayor sobre lo que sucede en nuestra aplicación, no es una opción válida.

En cuanto al precio que cuesta contratar este servicio, como se puede ver en la **IMAGEN 2** tiene distintas opciones según las características que se están buscando, y pensando en una empresa de tamaño pequeño o mediano, son precios muy asequibles, pero hay que recalcar el grado de personalización.

	Messages per day	Concurrent connections	Level of support	Monitoring integrations	Price	Annual price discount
Sandbox	200k	100	Standard	✗	Free	✗
Startup	1 million	500	Standard	✗	$49/month	✗
Pro	4 million	2,000	Standard	✓	$99/month	✗
Business	10 million	5,000	Premium	✓	$299/month	✗
Premium	20 million	10,000	Premium	✓	$499/month	✗
Growth	40 million	15,000	Premium	✓	$699/month	Get in touch
Plus	60 million	20,000	Premium	✓	$899/month	Get in touch
Growth Plus	90 million	30,000	Premium	✓	$1,199/month	Get in touch

IMAGEN 2 PUSHER: PRECIOS

QUICKBLOX

Se trata de un Software Development Kit (SDK) que funciona con el servidor propio del servicio. Internamente esta creado con los servicios de Google, es decir con Firebase. Para poder usar este servicio se debe instalar la librería en la aplicación cliente, configurar con las credenciales apropiadas de QuickBlox y añadir el código necesario que indican en su documentación. Funciona en varias plataformas.

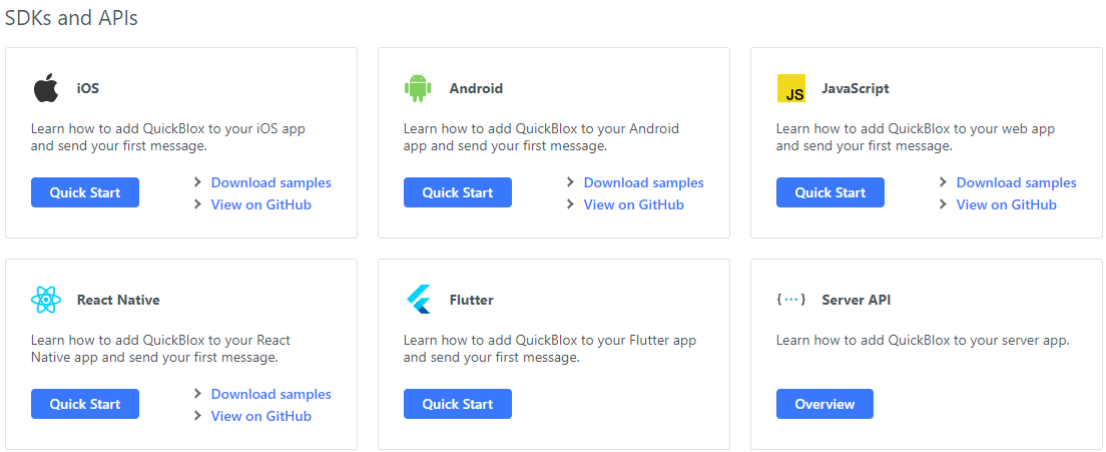


IMAGEN 3 QUICKBLOX: SOLUCIONES DISPONIBLES

En cuanto al precio para usar este servicio, nos ofrece varios planes incluso uno gratuito (IMAGEN 4). Se trata de un plan bastante sencillo, pero al menos no es versión sandbox, es decir una versión solo para hacer pruebas. El resto de los planes también son asequibles si se tratara de una empresa pequeña o mediana, pero puede suponer un problema para proyectos que se lanzan con muy bajo presupuesto.

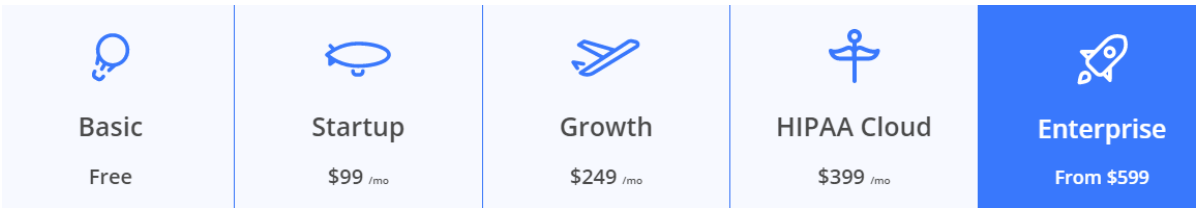


IMAGEN 4 QUICKBLOX: PRECIOS DE CONTRATACIÓN

STREAM

De la misma forma que la anterior nombrada, también se trata de un SDK el cual debe ser instalado en la aplicación y configurado con las credenciales propias del servicio. La documentación que ofrecen es bastante detallada y un aspecto importante es que está ordenada. Indican los pasos a seguir para desplegar correctamente el servicio e incluso incluyen un apartado de personalización. En cuanto a plataformas disponibles también dispone de una gran variedad.

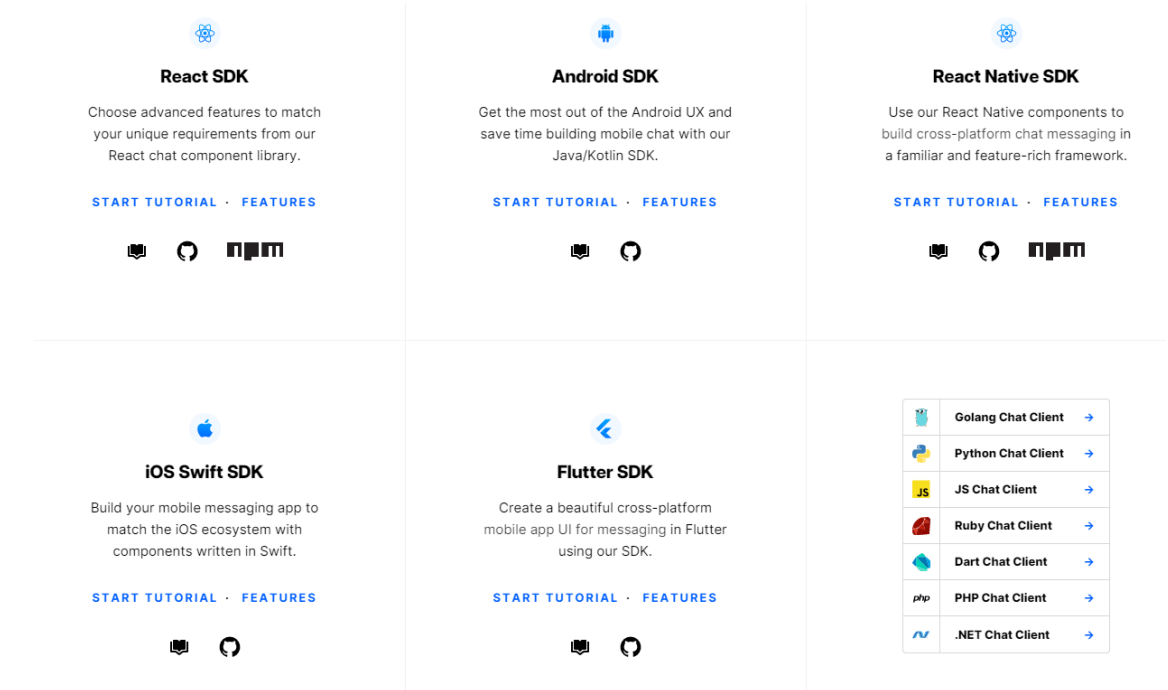


IMAGEN 5 STREAM: PLATAFORMAS DISPONIBLES

En cuanto a precios (IMAGEN 6), debe estar enfocado a otro tipo de mercados o países, porque el precio de la versión Startup, que correspondería con una empresa pequeña o mediana, es igual de elevado que las versiones Premium de los servicios comentados anteriormente. Es cierto que ofrecen más personalización, pero los precios son bastantes elevados. Hay que añadir que dispone de una prueba gratuita.

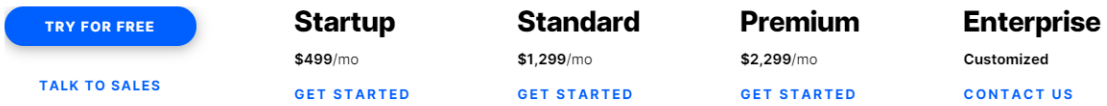


IMAGEN 6 STREAM: PLANES DE CONTRATACIÓN

CHAT21

Al analizar este servicio nos encontramos con el primer servicio de código abierto, lo que nos indica que al menos vamos a tener un extra de personalización, al poder modificar el código del servicio a nuestro gusto. Como la mayoría de los servicios anteriores, también se trata de un SDK e indican directamente que se trata de un chat basado en Firebase, por lo que vamos a tener 2 partes. Por un lado, vamos a tener que instalar el SDK en nuestra aplicación y por otro lado vamos a tener que crearnos una cuenta de Firebase, ya que sin este elemento no funcionaría el servicio, para conectarla con el SDK. En cuanto a plataformas, está disponible para las principales. Y sobre el precio o planes que ofrece, es gratuito.

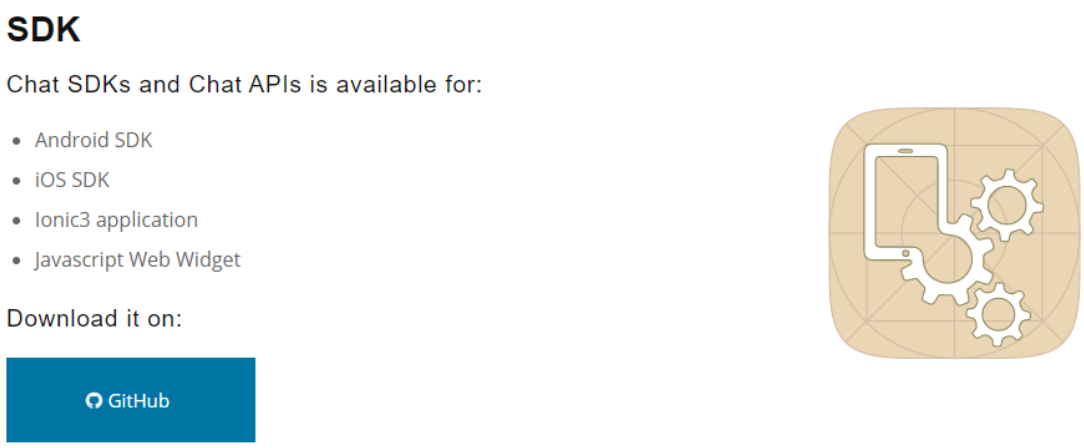


IMAGEN 7 CHAT21: PLANES DISPONIBLES

SENDBIRD

Nuevamente un SDK a instalar en nuestra aplicación y con el cual tendremos que realizar la conexión con los servidores del servicio. Mediante una detallada guía, siguiendo los pasos que indican, conseguiremos desplegar rápidamente el chat en la solución final. Además, ofrecen información para incorporar los elementos que se quiera al chat base. Es un servicio muy completo. En cuanto a plataformas disponibles también dispone de una gran variedad.

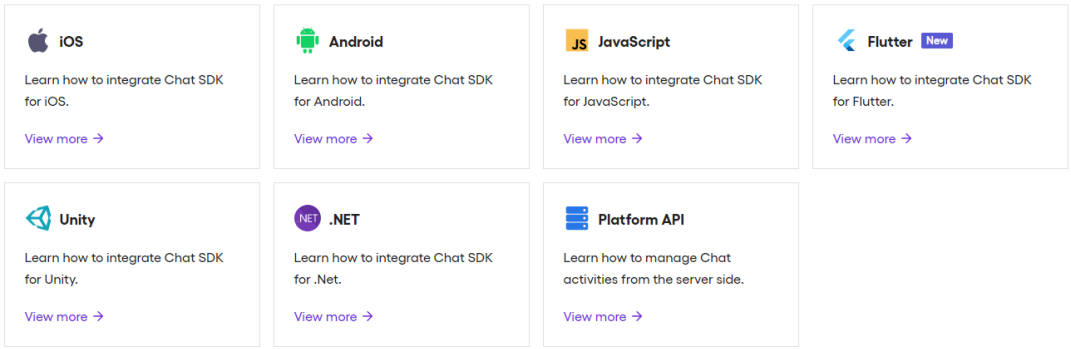


IMAGEN 8 SENDBIRD: PLATAFORMAS DISPONIBLES

En cuanto a precios, como se puede ver en la IMAGEN 9 ofrecen una versión de prueba, pero si se quiere contratar el servicio no va a ser nada barato. Es un caso como el del servicio **Stream** los planes que ofrece son bastante caros para una empresa pequeña o mediana, aunque en este caso, es algo más barato que Stream y parece que ofrece más características y de forma más clara.

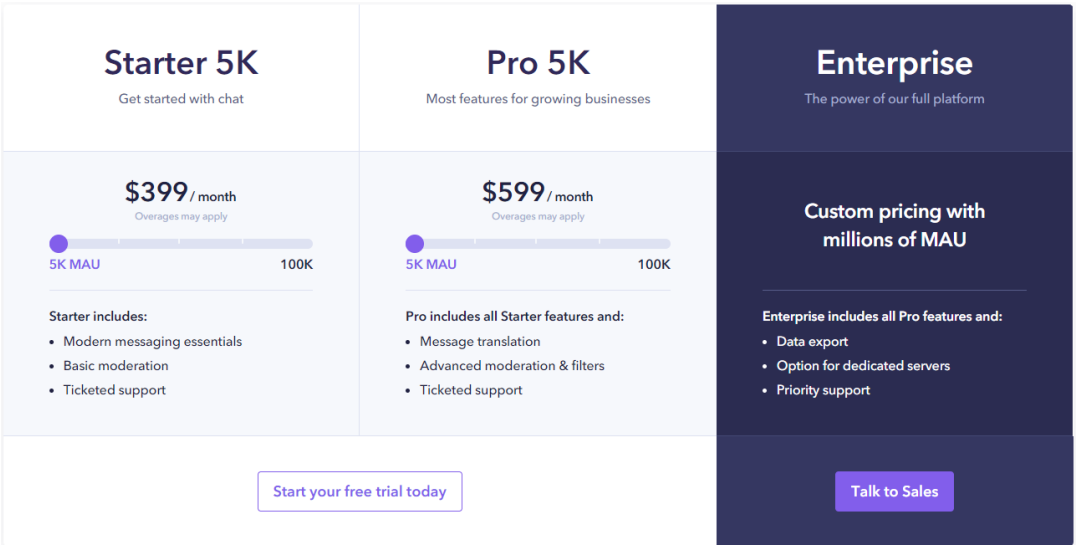


IMAGEN 9 SENDBIRD: PRECIOS DE CONTRATACIÓN

Como se ha visto a lo largo de este breve análisis, las opciones que tenemos en el mercado son muy similares entre sí. Casi todas se tratan de un SDK de pago, que se conectan con el servidor del propio servicio y gestionan las peticiones.

Esto puede parecer muy cómodo, pero tiene sus desventajas:

- Poco control o prácticamente nulo sobre la parte del servidor: Dependiendo del servicio se pueden revisar o modificar algunos parámetros, pero muchos de ellos son bastante cerrados. Es decir, se trata de sistemas cerrados, que nos dan soporte para el servicio que necesitamos, pero no podemos acceder y modificar el contenido de estos.

- Personalización escasa. Este aspecto va de la mano con el anterior: Al ser un sistema bastante cerrado, tenemos poca personalización a nivel de funcionamiento del sistema. Se va a tener que ceñir a la estructura de trabajo del servicio.

- Precio elevado: Dependiendo de la entidad que va a hacer uso del SDK, el esfuerzo económico sería reseñable. Además, muchas incorporan más funciones, no solo la del chat. Por lo que, si se pagara por algún servicio de este estilo, habría funciones que estaríamos pagando y que se necesitarían.

- Demasiada dependencia y problemas de fiabilidad: Al tratarse de un servicio externo a tu aplicación, es decir que tú no tienes control sobre ello, puedes tener algún problema como que el servicio falle mucho, o no funcione. Si el servicio del que se depende no está funcionando, se va a tener que esperar a que resuelvan el problema para volver a disponer del chat.

- Tecnología utilizada: Actualmente hay 3 grandes proveedores para servicios Cloud: Amazon Web Services (AWS), Firebase de Google y Azure de Microsoft. Aunque la opción de Firebase es la que menos cuota de mercado tiene, es la más utilizada en este tipo de SDK. Lo que significa que muchas aplicaciones o servicios que trabajan con AWS o Firebase, no pueden utilizar este tipo de librerías fácilmente.

- Poca flexibilidad: Al estar tan definidos todos los elementos, es difícil reemplazar alguno o modificarlo, por ejemplo, algo tan sencillo como los parámetros que se quiere que lleve el mensaje. Es decir, falta de MVC.

Contrarrestando estas desventajas, nuestra librería es capaz de brindar toda esa flexibilidad y personalización que no nos ofrecen las demás:

- Está basada en Azure de Microsoft, lo que significa que se está trabajando con uno de los mejores y más utilizados servicios Cloud.

- A pesar de estar construida en Azure, se podría cambiar el servicio Cloud gracias a su estructura MVC y a los scripts que crean la estructura de la base de datos.

- Se tiene un control total sobre los datos. Se pueden visualizar, modificar los datos y las estructuras según las necesidades.

  La comunicación entre aplicación y servidor se realiza directamente, por lo que supone una ventaja para el funcionamiento integral de la aplicación como tanto para la protección de datos.

- Contiene únicamente la función de chat, que es lo que se busca en la mayoría de los casos, sin tener que pagar por funcionalidades extras que no se necesitan.

- Se puede personalizar completamente.

En resumen, utilizar la librería desarrollada supone una ventaja de personalización, control, fiabilidad y flexibilidad a la hora de implementar una nueva aplicación Android y en el futuro iOS.

Análisis

Metodología

Para el desarrollo de la librería se ha usado una combinación de dos metodologías: cascada y la incremental.

FUNCIONAMIENTO DE LA METODOLOGÍA EN CASCADA

Se desarrollan las diferentes funciones en etapas diferenciadas y obedeciendo un riguroso orden. Antes de cada etapa se debe revisar el producto para ver si está listo para pasar a la siguiente fase. Los requisitos y especificaciones iniciales no están predispuestos para cambiarse, por lo que no se puede ver los resultados hasta que el proyecto ya esté bastante avanzado.

FUNCIONAMIENTO DE LA METODOLOGÍA INCREMENTAL

Se va construyendo el producto final de manera progresiva. En cada etapa incremental se agrega una nueva funcionalidad, lo que permite ver resultados de una forma más rápida en comparación con el modelo en cascada. El software se puede empezar a utilizar incluso antes de que se complete totalmente y, en general, es mucho más flexible que las demás metodologías.

En este proyecto, al principio se utilizó la metodología cascada, con la cual se estudió que se debía hacer, qué debía contener y cómo se quería hacer. Una vez se definieron los requisitos, se dividieron por funcionalidades, se diseñaron y se empezó a implementar. De esta forma se estaba cambiando la metodología a incremental, lo que supone que a medida que se vayan implementado los requisitos correspondientes a cada funcionalidad, ya se tendrá una herramienta funcional que con el tiempo tendrá más opciones (IMAGEN 10).

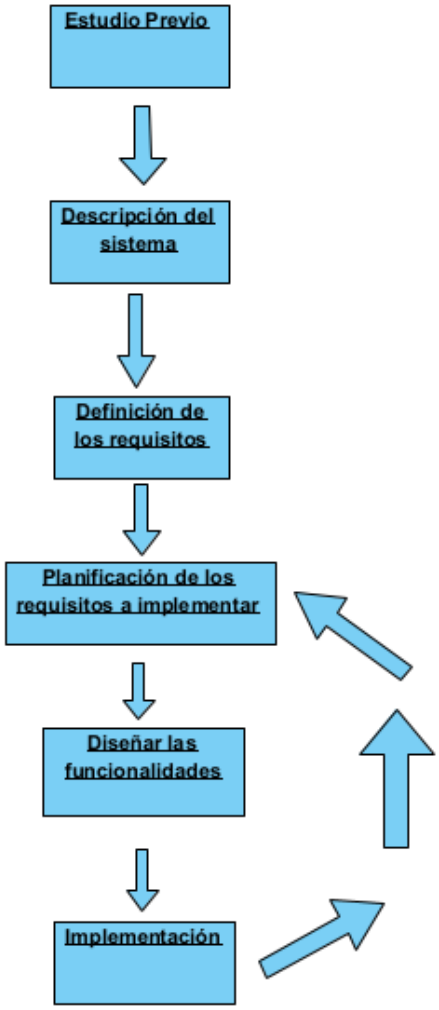


IMAGEN 10 METODOLOGÍA DEL PROYECTO

Así pues, una vez que se planifiquen los requisitos a implementar y se diseñen, se irán implementando las distintas funciones progresivamente y cuando se entre en dicho bucle de planificar, diseñar e implementar, es cuando se cambia la metodología a incremental.

Organización

En cuanto a la organización que se ha seguido para el desarrollo de este proyecto, como se ha comentado antes, se ha basado en funcionalidades y agrupaciones de requisitos. Para ello, se han establecido reuniones con el tutor del proyecto cada dos semanas, para revisar el estado del proyecto.

Para definir los objetivos de cada reunión, se ha utilizado una herramienta simple en Excel, diseñada por el tutor:

Fechas				Progreso:		
Lunes - Domingo				100%		
Horario						
Flexible						
Responsable						
Garbiel Oniga						
Comentarios						
Proyecto	**Tarea**	**Horas planificadas**	**Horas reales realizadas**	**Terminado**	**Validado (Iván)**	**Comentarios**
	Generar APK	0,5	0,05	☑	OK	
	Cuando estás viendo el Chat (conversación) con una persona y llega una Push, actualizar el listado de mensjes	1	0,1	☑	OK	
	Revisar RQ marcados en naranja y revisar los que ya están hechos	0,5	0,15	☑		RQ4, RQ5, aun no estan probados. RQ7, falta que cuando el mensaje no esté leido aparezca en negrita, pero depende de los requisitos de estado del mensaje. RQ12, RQ13 revisar implementación, porque parece que se muestran todos los mensajes.
	Revisar código servidor	2	1,5	☑		
	Diseñar el módulo de conversaciones en grupo para Android	4	4	☑		
	Diseñar el módulo de conversaciones en grupo para Servidor	4	1,5	☑		
	Reunión	2	1	☑		
	Generar la pantalla para conversación en grupo	4	2	☑		
				☐		
				☐		
				☐		
	Total	18	10,3			
		Porcentaje de finalización:				
		100,00%				
		0,00%				
		100,00%				

IMAGEN 11 ORGANIZACIÓN DEL PROYECTO

Se trata de una tabla donde se especifican las tareas que se deben cumplir, el tiempo estimado en el que se debería poder solventar dicha tarea, el tiempo real que ha costado hasta finalizar la tarea y finalmente si está completada y validada por el tutor.

Todo esto va asociado a un porcentaje, con el cual vamos a tener información de como de avanzadas tenemos las tareas para estas semanas, si estamos a punto de terminarlas o si nos falta mucho trabajo y de esta manera ser constantes con el trabajo realizado. Muchas veces no se conseguía realizar el 100% de las tareas, por lo que para la siguiente sesión se volvía a anotar en la lista de tareas.

Por último, decir que, aunque esta solución de organización mediante Excel no sea la más profesional, es mucho más intuitiva y fácil de utilizar que otras.

Casos de Uso

Para determinar las acciones que se pueden realizar en nuestra librería, se van a presentar mediante los casos de uso. Concretamente en tres escenarios, por parte del usuario, la aplicación y por parte del servidor.

USUARIO

CUU1. Cambiar de pestaña: el usuario podrá cambiar de pestañas libremente para visualizar conversaciones, contactos o grupos.

CUU2. Seleccionar conversación: se podrá seleccionar la conversación a la que se quiere acceder.

CUU3. Seleccionar contacto: se podrá seleccionar un contacto con el cual iniciar una conversación.

CUU4. Seleccionar grupo: el usuario podrá seleccionar el grupo del cual quiera visualizar la conversación.

CUU5. Enviar mensaje: se podrá enviar mensajes tanto individuales como grupales.

CUU6. Borrar conversación individual: el usuario tiene la capacidad de borrar una conversación individual si quiere.

CUU7. Ver Mensajes: el usuario podrá ver los mensajes de las conversaciones tanto individuales como de grupo.

CUU8. Cambiar su estado: se podrá cambiar el estado a Disponible u Ocupado.

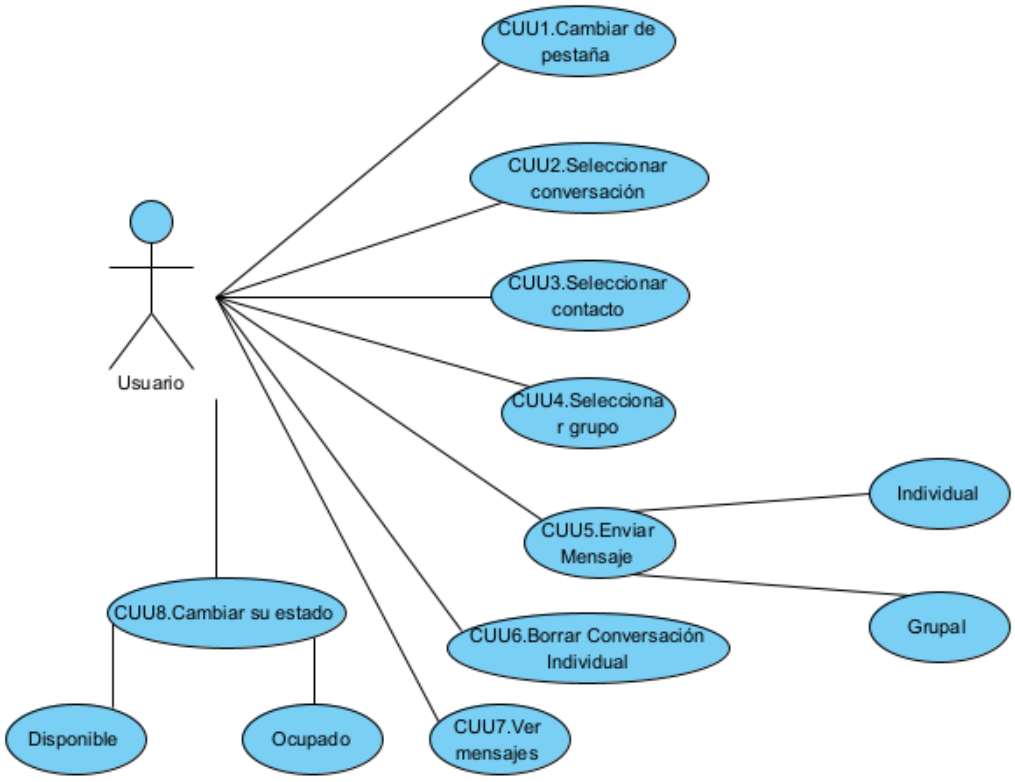


IMAGEN 12 CASOS DE USO DEL USUARIO

LIBRERÍA

CUL1. Obtener contactos: la librería obtendrá los contactos disponibles.

CUL2. Recibir mensajes: se recibirán mensajes nuevos.

CUL3. Mostar notificación: en caso de recibir mensajes nuevos, se mostrará una notificación.

CUL4. Actualizar lista de conversaciones: Cuando se cree una conversación nueva o llegue un nuevo mensaje se actualizará. Además, si se está viendo la lista de conversaciones y nos llega un mensaje nuevo, se actualizará en tiempo real.

CUL5. Actualizar lista de mensajes dentro de una conversación: cuando llegue un nuevo mensaje se actualizará la lista de mensajes correspondiente a la conversación. Además, si nos encontramos en la conversación y nos llega un mensaje nuevo, se actualizará en tiempo real.

CUL6. Obtener mensajes: la librería obtendrá todos los mensajes.

CUL7. Comprobar conversación leída: antes de mostrar la lista de conversaciones, se comprobará si la hemos leído o no.

CUL8. Cambiar estado de una conversación: en caso de que hayamos leído la conversación, se cambiará el estado de esta.

CUL9. Obtener grupos: la librería podrá obtener los grupos a los que pertenecemos.

CUL10. Mostar lista de mensajes: se mostrará el listado de mensajes en caso de abrir una conversación.

CUL11. Mostar lista de contactos: mostrará el listado de contactos al acceder a la pantalla correspondiente.

CUL12. Mostrar lista de conversaciones: mostrará la lista de conversaciones que se hayan iniciado cuando estemos en la pantalla correspondiente.

CUL13. Mostrar lista de grupos: mostrará la lista de grupos que se hayan iniciado cuando estemos en la pantalla correspondiente.

CUL14.Crear conversación: cuando el usuario envíe o reciba un mensaje de un destinatario nuevo, la librería creará una conversación.

CUL15. Cambiar estado: la librería podrá notificar al servidor de que el usuario quiere cambiar su estado.

IMAGEN 13 CASOS DE USO DE LA LIBRERÍA

SERVIDOR

CUS1. Recibir mensaje: el servidor puede recibir un mensaje por parte de el/los clientes y puede ser tanto un mensaje individual como grupal.

CUS2. Enviar mensaje: cuando el servidor reciba una petición de envío de mensaje, podrá hacerlo, tanto a usuarios individuales como a grupos.

CUS3. Notificar mensaje nuevo: cuando un usuario o grupo reciba un nuevo mensaje, la API notificará al servicio de notificaciones (Notification Hubs) que debe mostrar dicha notificación.

CUS4. Devolver lista mensajes: el servidor devolverá la lista de mensajes para cada conversación.

CUS5. Devolver lista grupos: se devolverá la lista de los grupos de los cuales forma parte el usuario.

CUS6. Gestionar usuarios de un grupo: se podrá gestionar la participación de los usuarios en uno o varios grupos.

CUS7. Cambiar estado del usuario: el servidor podrá cambiar el estado dentro de la BD de un usuario

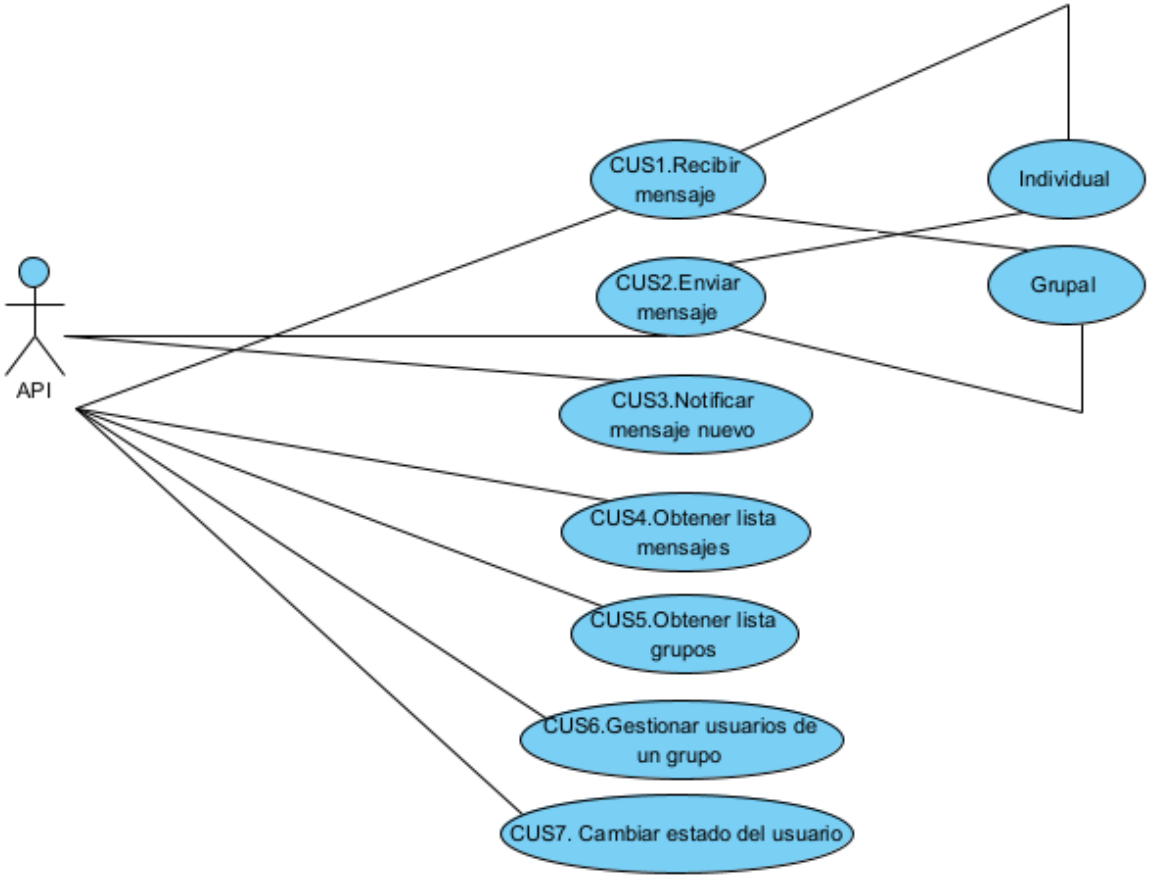


IMAGEN 14 CASOS DE USO DEL SERVIDOR

Requisitos

A continuación, se van a determinar los requisitos que debe cumplir la librería, se encontraran divididos en varias secciones, que coinciden con cada elemento de la librería. Asimismo, habrá dos tipos de requisitos, funcionales (que deben ser implementados y estar operativos correctamente) y los no funcionales (que no son directamente testeables).

REQUISITOS FUNCIONALES

RQF1. Se podrán realizar conversaciones individuales. Lo que significa que habrá intercambio de mensajes entre un usuario A y un usuario B.

RQF2. Asimismo, se podrán realizar conversaciones grupales, siendo estas el intercambio de mensajes entre N usuarios.

LISTADO DE CONVERSACIONES

RQF3. Se tendrá un listado de conversaciones que contendrá tanto las individuales como las grupales.

RQF4. En el listado de conversaciones, aparecerán todas aquellas que contengan al menos un mensaje.

RQF5. El listado estará ordenado cronológicamente estando en primer lugar las conversaciones más recientes y en último las más antiguas.

RQF6. La información que se tendrá disponible al ver el listado es el nombre del usuario con el que estamos teniendo la conversación en caso de tratarse de una conversación individual y el nombre del grupo en caso de una grupal. En ambos casos se mostrará la foto, el último mensaje de la conversación y el estado de este.

RQF7. Por defecto, la lista mostrará todas las conversaciones disponibles.

FQF8. La pantalla que mostrará el listado de conversaciones estará pendiente de recibir nuevos mensajes para actualizar la lista de mensajes.

RQF9. Cuando se reciba un nuevo mensaje, se actualizará la pantalla que muestra las conversaciones automáticamente con los nuevos datos.

RQF10. Cuando se actualice el listado de conversaciones, este se volverá a ordenar cronológicamente.

CONVERSACIÓN

RQF11. Se podrá ver el listado de mensajes que se han intercambiado entre un usuario A y un usuario B en caso de conversaciones individuales y todos los mensajes en caso de Chat grupales.

RQF12. Dichos mensajes se mostrarán de manera cronológica, estando los más recientes al final de la lista y los más antiguos al principio.

RQF13. Al mostrar el listado de mensajes, cuando el día se diferente, se mostrará un indicativo del día al que pertenecen los mensajes.

RQF14. Por defecto se cargarán todos los mensajes de los que disponga la conversación.

RQF15. La pantalla que mostrará los mensajes de una conversación estará pendiente de recibir nuevos mensajes para actualizar la lista de mensajes.

RQF16. Cuando se reciba un nuevo mensaje, el evento actualizará la pantalla que muestra los mensajes con los nuevos datos.

RQF17. Los elementos que contendrá la pantalla de una conversación serán los siguientes:

RQF17.1. Una caja de texto en la cual se deberá escribir el mensaje que se desea enviar.

RQF17.2. Si esta caja de texto está vacía y se desea enviar un mensaje, no se realizará ninguna acción.

RQF17.3. Contendrá también un botón cuya funcionalidad será enviar el mensaje.

RQF17.4. Si a la hora de enviar el mensaje, ocurre algún error, el mensaje no se añadirá a la lista.

RQF17.5. Cuando se produzca un error de envío, se notificará con un mensaje emergente.

RQF17.6. En la parte superior de la conversación estará el nombre del usuario o grupo con el que se está intercambiando mensajes.

RQF17.7. En la parte izquierda del nombre, se mostrará la foto del contacto o grupo.

RQF17.8. Finalmente, a la izquierda de la foto, habrá un botón con icono de flecha, la cual volverá a la pantalla de conversaciones en caso de ser pulsado.

LISTADO DE CONTACTOS

RQF18. Se mostrará un listado con todos los contactos individuales del que dispone el usuario.

RQF19. De los contactos se mostrará el nombre, la foto y su estado, si están disponibles u ocupados.

RQF20. En caso de pulsar sobre alguno de los contactos, se abrirá la conversación.

RQF21. Si no estaba creada la conversación, es decir no se han intercambiado ningún mensaje, en caso de que se haga, se creará la conversación y el mensaje se añadirá a la lista.

RQF22. En caso de que no se realice ningún envío de mensaje, al salir de la conversación no pasará nada.

LISTADO DE GRUPOS

RQF23. Se mostrará una lista con los grupos a los que el usuario pertenece.

RQF24. De los grupos se mostrará el nombre, la foto y como información adicional se mostrará la fecha en la que se creó.

RQF25. En caso de pulsar sobre alguno, se mostrará la conversación de grupo.

GRUPOS

RQF26. La modalidad de creación de grupos (cualquier usuario puede crear un grupo o solo usuarios administradores pueden crear grupos), dependerá de la aplicación final que implemente la librería.

RQF27. Tendrán una capacidad que estará determinada por la solución final.

RQF38. En caso de que envíen mensajes más de un usuario a la vez, estos serán ordenados de manera cronológica.

ESTADO DE LAS CONVERSACIONES

RQF29. Cuando se cree una nueva conversación, el estado por defecto será no leída.

RQF30. Cuando el usuario lea la conversación, esta cambiará de estado a leída.

RQF31. El check del estado de la conversación de color negro significa no leída y el azul, leída.

NOTIFICACIONES PUSH

RQF32. Estando la aplicación cerrada, se deberán recibir las notificaciones que avisen de los nuevos mensajes.

REQUISITOS NO FUNCIONALES

RQNF1. La librería se podrá usar en dispositivos Android con versión Android superior a 26 (Android Oreo), los servidores programados en .NET con una mínima versión 2.2.4.

RQNF2. En los dispositivos móviles, se guardarán copias locales completas de la base de datos de los mensajes pertenecientes al usuario siempre que se pueda.

RQNF3. Los mensajes deberán ser guardados de forma segura para proteger los datos.

Arquitectura

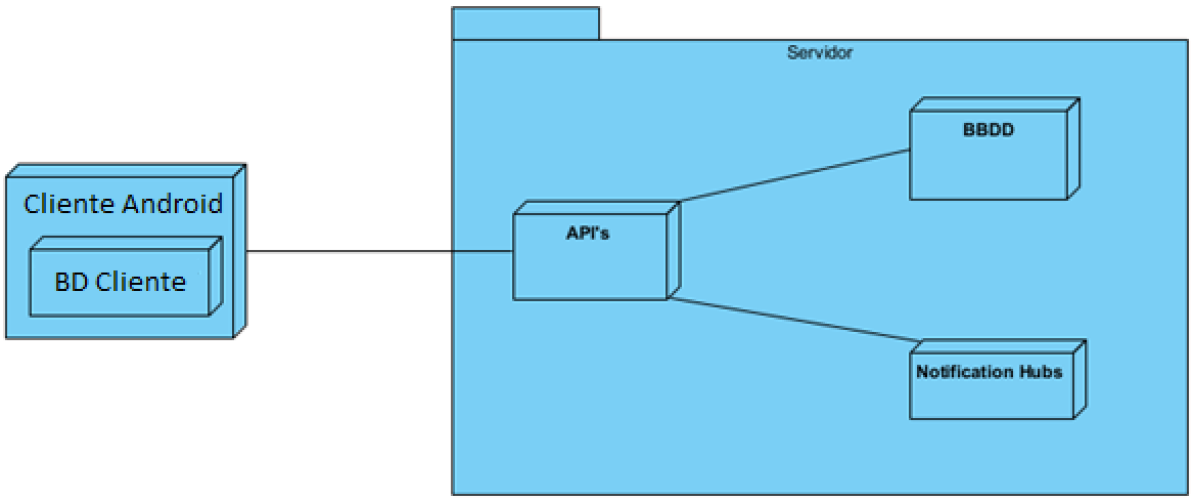


IMAGEN 15 ARQUITECTURA DEL SERVICIO

A continuación, se va a explicar cómo es la estructura de la librería, las conexiones que tiene y de qué forma se comunica.

CLIENTE ANDROID

El cliente Android, desde la cual se van a enviar los mensajes al servidor, o donde llegaran los nuevos. Para comunicarse con el servidor, lo hace mediante llamadas a Application Programming Interfaces (API's). Para cada función específica hay una API que trata la solicitud del cliente y le devuelve una respuesta.

Por otro lado, el cliente tiene una BD interna en la cual tiene una réplica de la estructura de la BD del servidor y donde almacena los mensajes enviados y recibidos.

SERVIDOR

Como se puede ver en la anterior **IMAGEN 15**, en el servidor hay 3 elementos importantes: por un lado, las API's que son las que se comunican con el cliente, la BD de la librería y el notification hubs que es el encargado de enviar el aviso a los usuarios de que tienen nuevos mensajes.

API'S

Como se ha comentado antes, las API's son el canal de comunicación entre el cliente y el servidor. Hay varias y cada una tiene su función específica, como por ejemplo comprobar si el usuario tiene nuevos mensajes, o saber a cuantos grupos pertenece. Además, son estas las que acceden a la BD para obtener información y notificar al HUB de que hay nuevos mensajes para que muestre la notificación push.

BASE DE DATOS

Se trata de la base de datos, es donde se almacenan todos los datos necesarios para el funcionamiento de la librería y a la cual se accede mediante consultas desde las API's.

NOTIFICATION HUBS

Es el encargado de enviar notificaciones Push a los dispositivos suscritos a los eventos. Una API realiza una llamada a este servicio, que se encarga de enviar la notificación push correspondiente. Con una sola llamada, es capaz de enviar las notificaciones a dispositivos Android, iOS u otras opciones como Windows, Kindle…. Está diseñado para escalados de gran volumen y, además, funciona con cualquier back-end lo que lo convierte en muy versátil. Usar un servicio como este, facilita mucho el uso de notificaciones, ya que todo el funcionamiento se concentra en un Hub que gestiona todo el tráfico.

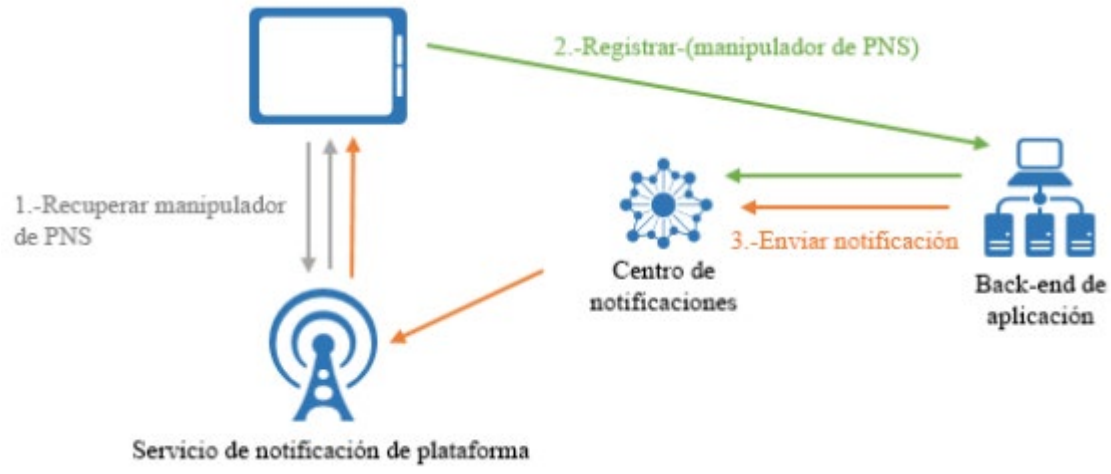


IMAGEN 16 FUNCIONAMIENTO NOTIFICACION HUBS

Tecnologías

Para realizar este proyecto, se han utilizado distintas tecnologías, algunas para la parte del cliente, otras para la parte del servidor y finalmente otras para la documentación u otros. A continuación, se van a enumerar y a describir brevemente:

- Android Studio: ha sido el programa con el que se ha desarrollado la aplicación Android

- Visual Studio: para poder desarrollar el servidor con todos sus elementos, se ha realizado con este programa, ya que se puede conectar directamente con Azure (que ha sido la plataforma cloud que se ha elegido) y trabajar cómodamente tanto con las API's como con las BBDD.

- Azure: como se acaba de comentar, de entre las opciones que había para desarrollar nuestra librería, se ha escogido Azure porque es una de las más utilizadas y eso supone varias ventajas. Por un lado, el funcionamiento en sí, ya que, al tratarse de una de las grandes, sus infraestructuras son fiables.
  Por otro lado, cuando se quiere incorporar una nueva función en una aplicación, en este caso nuestra librería de chat, vamos a querer tener lo mejor, y utilizando Azure es lo que tenemos.

- SQL Server: se podía haber utilizado otro tipo de gestor de BBDD, pero en este caso como se está trabajando con el entorno de Microsoft, más concretamente de Azure, se ha querido unificar todo. Además, es una de las opciones que nos ofrece Azure a la hora de crear una BD.

- SQLite: se ha utilizado para poder crear una BD en el dispositivo móvil, y ser capaces de almacenar los mensajes, contactos y grupos. De esta forma, se tienen los datos más accesibles y a cambio de menos recursos a la hora de obtenerlos.

- Postman: para poder controlar y comprobar el funcionamiento de nuestras API's se ha utilizado esta herramienta. Es un programa que permite realizar llamadas a API y ver el resultado que nos devuelve, así podemos hacer pruebas más cómodamente.

- Bitbucket: una de las muchas herramientas basadas en GIT, se ha utilizado para el control de versiones, cuando se lograba implementar una nueva funcionalidad se guardaba para que si en el futuro había algún problema tener una copia de seguridad. Se ha utilizado tanto para el cliente como para el servidor.

- KeePass: un gestor de contraseñas para almacenar de forma segura las contraseñas de acceso a la plataforma Azure, a su BBDD y a demás elementos.

- Visual Paradigm: se ha utilizado este programa para la creación de la documentación, más en específico para todos los diagramas.

- Balsamiq: es un programa con el cual se pueden realizar maquetas de las interfaces, por lo tanto, se ha utilizado para crear los bocetos de las interfaces de las que dispone nuestro cliente.

Lenguaje de Programación

Para definir los lenguajes de programación que se han utilizado en el proyecto, se van a separar en dos grupos, por un lado, los utilizados para el cliente, y por otro los utilizados para el servidor, ambos acompañados de una breve explicación del porqué se ha utilizado estas tecnologías y no otras.

Cliente Android

- Java: las aplicaciones Android pueden ser desarrolladas tanto en Java como el Kotlin, pero para este caso, se ha elegido Java por varias razones:

    - Aunque Kotlin se creó con el propósito de mejorar Java, mejorando los puntos más débiles que tiene, es un lenguaje joven y aún está en fase de crecimiento.
    - Como es un lenguaje joven, aun no hay toda la información que se dispone con Java para crear proyectos que interconecta tecnologías específicas como nuestra librería.
    - A pesar de que la cuota de mercado de Kotlin está aumentando, Java se sigue utilizando muchísimo y puede ser una moda pasajera.

- SQL: se utiliza para las consultas internas de la BD de Android, para crear las tablas, almacenar y consultar datos.

Servidor azure

El servidor está basado en Azure y para ello se ha utilizado el entorno de trabajo (framework) de .NET, que aparte de Azure puede trabajar con muchas otras tecnologías.

- C#: es el lenguaje de programación base

- .NET: es un framework de Microsoft que hace un énfasis en la transparencia de redes, con independencia de plataforma de hardware y que permite un rápido desarrollo de aplicaciones.

- Entity Framework: trabajos con la base de datos, como los modelos de las clases, consultas…

- Linq: consultas con la base de datos

Interfaces de Usuario

A continuación, se van a adjuntar y describir brevemente los bocetos que se hicieron para definir el aspecto de la interfaz en la aplicación. Son bocetos simples, ya que, al tratarse de una librería personalizable, el aspecto final será dado por la solución definitiva.

Se tratan de 3 pantallas, una para las conversaciones, tanto individuales como grupales, una para los contactos disponibles y una tercera para los grupos. Este diseño se ha planteado de esta manera, que en una primera versión de la librería, la creación de los grupos así como su gestión, será controlado de manera manual por un administrador con acceso al servidor. Además, está la posibilidad de que se quiera tener únicamente conversaciones individuales, o, por el contrario, solo grupales.

Chats

En la parte de abajo tenemos el menú de navegación, y en la parte principal de la pantalla se encuentra la lista de conversaciones con los que se ha intercambiado mensajes alguna vez. Además, para cada conversación se mostrará la foto, fecha del último mensaje, además del estado de la conversación, lo que se determinará el color del check. (**Anexo 1**)

Contactos

En cuanto a la pantalla de contactos, se visualizarán todos los contactos que el usuario tenga y con los cuales puede iniciar una conversación en caso de que no la haya tenido anteriormente, o visualizar la conversación actual. Para cada conversación se mostrará la foto del contacto, y un dato TBD como por ejemplo un apodo del usuario. (**Anexo 2**)

Grupos

Finalmente, en cuanto a la pantalla de chats grupales, en caso de que el usuario se encuentre en uno o más grupos, se mostrará un listado de estos junto a la fecha de creación de estos. (**Anexo 3**)

Diseño e Implementación

Para explicar cómo ha sido el proceso de implementación de la librería de chat integral, se va a dividir en tres secciones:

Servidor

Dentro del servidor hay unas partes clave en el funcionamiento de la herramienta, la parte más directa con la que se conecta el cliente, son los controladores, es decir las API's. Y después hay otros elementos de los que hace uso las API's para su correcto funcionamiento.

API's

UniversalChatController

Es el encargado de toda la lógica de mensajes, es decir de enviar los mensajes y de obtenerlos.

- HttpGet("GetAllNewMessages"): Con este método de tipo GET, obtenemos todos los mensajes en los cuales está involucrado el usuario. Pertenecientes tanto a conversaciones individuales como grupales. (ANEXO 4)

- HttpPost("SendChatMessage"): Y con este método tipo POST, cuando se envíe un mensaje, en caso de estar todo correcto, se almacenará en la base de datos. (ANEXO 5)

ChatGroupsController

Como su propio nombre indica, a través de esta API obtendremos los grupos a los que pertenece el usuario.

- HttpGet("GetAllGroupsByMemberEnrolled"): Para obtener los grupos a los que pertenece el usuario, se utilizará este método GET, el cual devolverá una lista de todos los grupos y con los cuales podremos filtrar los mensajes para mostrarlos posteriormente. (ANEXO 6)

MembersEnrroledController

Con este controlador, se puede administrar toda la información correspondiente con los miembros de la librería.

- HttpPost("ChangeUserStatus"): Más concretamente, nosotros utilizamos la API ChangeUserStatus, con la cual el usuario podrá establecer su estado en Disponible u Ocupado. (ANEXO 7)

INTERFACES

Para tener un buen diseño también se han hecho uso de las interfaces, las cuales determinan las funcionalidades que tenemos en nuestra librería.

IMessage

En esta interfaz se encontrarán todos los métodos que tengan que ver con la trata de mensajes.

```
Task<IEnumerable<Message>> GetAllNewMessages(Guid UserId, DateTime LastUpdateUTCInServer);
1 referencia
Task<bool> SaveUniversalMessageInServerDatabase(Message chatMessage);
```

IMAGEN 17 INTERFACES DE LA CLASE IMESSAGE

IChatGroup

Análogo a la interfaz anterior, contendrá todos los métodos que tengan que ver con los grupos.

```
Task<IEnumerable<ChatGroup>> GetAllChatGroupsByMemberEnrolled(Guid idMemberEnrolled);
```

IMAGEN 18 INTERFAZ DE LA CLASE ICHATGROUP

IMemberEnrolled

Interfaz que determina las API's disponibles relacionadas con los datos del usuario.

```
Task<bool> ChangeUserStatus(Guid idMemberEnrolled, String status);
```

IMAGEN 19 INTERFAZ DE LA CLASE IMEMBERENROLLED

CONCRETE

Continuando con los elementos que contiene el servidor para su correcto funcionamiento, nos encontramos con los Concrete, que son las clases que implementan las interfaces.

MessageConcrete

- GetAllNewMessages: Como se observa en el fragmento de código (**ANEXO 8**), primero de todo se obtienen el identificador de los grupos a los cuales pertenece el usuario. A continuación, se obtienen todos los mensajes de las conversaciones individuales y finalmente se obtienen los mensajes de las conversaciones grupales. Se concatenan y devuelven en forma de lista de mensajes.

- SaveMessageInServerDatabase: Con este otro método (**ANEXO 9**), cuando se envíe un mensaje, si se realiza el proceso de forma correcta, se almacenará en la base de datos.

ChatGroupConcrete

GetAllChatGroupsByMemberEnrolled: Como se ha comentado anteriormente, en cuanto a grupos el método que estamos tratando es el que obtener la lista de grupos en los cuales está involucrado el usuario. (**ANEXO 10**)

DTO

Una vez se tiene clara la estructura del servidor, se va a explicar cómo se envían y reciben los datos entre el cliente y el servidor. Para ello, y según nos indica la documentación de Azure, se deben definir unas clases de tipo Data Transference Object (DTO). Estas clases, definen un objeto que debe tener la misma estructura tanto en el servidor como el cliente para que la transferencia sea correcta. Realmente lo que se está realizando es un mapeo de las características de los mensajes o de los grupos para no trabajar directamente con las entidades de la base de datos, y también porque si se quiere realizar algún cambio con la cantidad de parámetros que se quieren de alguna clase, no tener que cambiar la entidad de la BD, solo se modifica el DTO.

ChatMessageDTOUniversal

Determina los parámetros que se utilizan en la transferencia al cliente. (**ANEXO 11**)

ChatGroupDTO

Determina los elementos que se utilizan de los grupos. (**ANEXO 12**)

MemberEnrolledDTO

Determina la estructura de los usuarios como se puede ver en ANEXO 13.

Una vez repasados todos los elementos del servidor, sin tener en cuenta las entidades de la base de datos, la estructura quedaría de la siguiente manera:

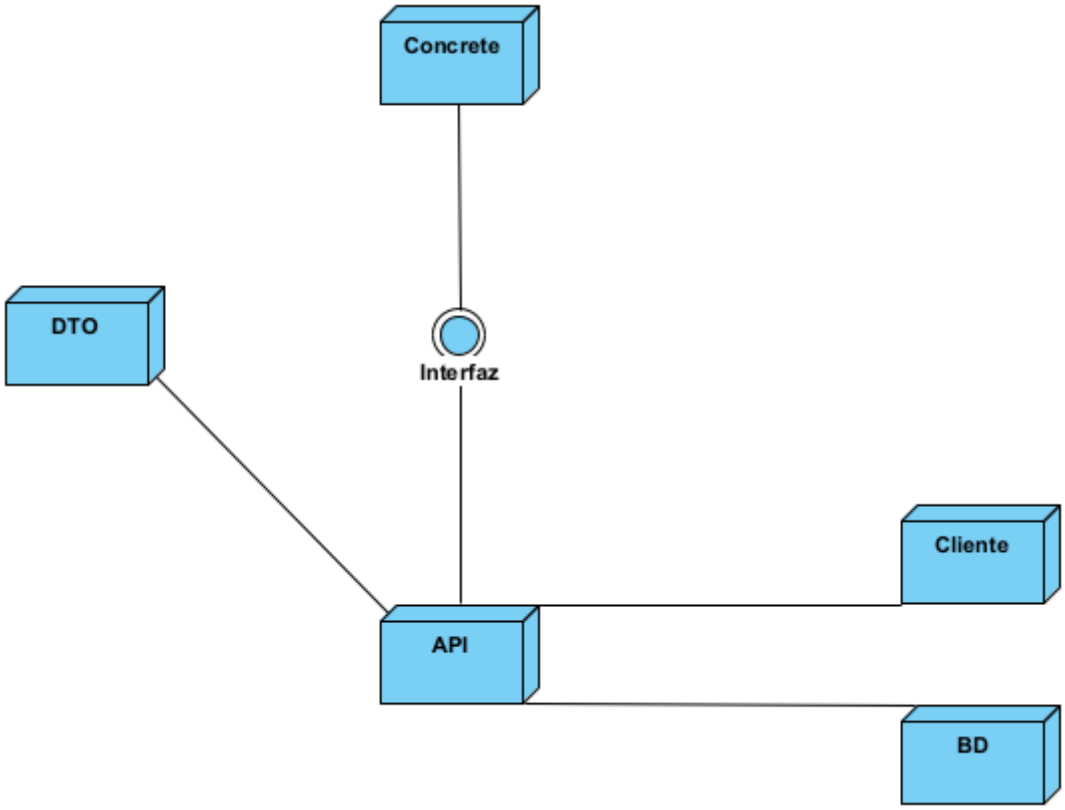


IMAGEN 20 DIAGRAMA DE DESPLIEGUE DE COMUNICACIONES

Las API's usan los métodos de los Concrete a través de las Interfaces. Después utilizando los DTO correspondientes, se comunican con el cliente y la BD.

Cliente

En cuanto al cliente, también contamos con una estructura similar a la del servidor, tiene interfaces, clases que especifican las llamadas a las API's, clases que interaccionan con la base de datos interna de la librería para almacenar la información del usuario y todas las clases relacionadas con la vista y con el comportamiento de estas. Se van a destacar unas cuantas clases cuya relevancia es superior.

SERVERCONNECTION

Esta clase es la encargada de hacer las llamadas a las API's del servidor. De la misma forma que anteriormente, se analizarán las más importantes.

sendChatMessage

En caso de que nos devuelva una respuesta favorable, se recogerá el resultado y se evaluará, en este caso si nos devuelve un "Ok", se actualizará la conversación y el mensaje se verá reflejado en la lista. (ANEXO 14)

getAllNewChatMessages

A la hora de obtener todos los mensajes (ANEXO 15), en caso de obtener un resultado favorable, el servidor nos mandará la lista con todos los mensajes. Se van a evaluar y se van a asociar a cada conversación correspondiente.

changeUserStatus

Para que el usuario pueda cambiar su estado de disponibilidad a Disponible u Ocupado, se ha implementado este método que llama a la API correspondiente en el servidor y que hace el cambio en la BD para que pueda verse en cambio para todas las personas. (ANEXO 16)

NOTIFICACIONES

Cuando el usuario recibe nuevos mensajes, gracias a una funcionalidad incorporada de Android Studio, heredada de Firebase, la librería lo detectará y procederá a realizar varias acciones.

La acción base, es obtener los nuevos mensajes, y eso se realiza de la misma manera que se ha explicado antes, mediante la clase ServerConnection. Si esta acción de obtener los nuevos mensajes se realiza correctamente se procederá a realizar las siguientes acciones:

1. Se creará una notificación push
2. Se comprobará la conversación o conversaciones y se añadirán a la lista los nuevos mensajes.
3. Se enviará la notificación y se mostrará
4. Se actualizarán las vistas de la lista de conversaciones y las conversaciones en sí.

Base de Datos

Como se ha mencionado brevemente antes, el servidor se comunica con la base de datos a través de entidades. Estas entidades representan la estructura que tiene internamente la BD y de esta forma se pueden almacenar, obtener o modificar elementos de la base de datos de forma más cómoda en objetos.

Un usuario puede tener varios mensajes y pertenecer a varios grupos, por lo que se debe crear una tabla Usuario_Grupo con la cual establecer la relación de los diferentes grupos.

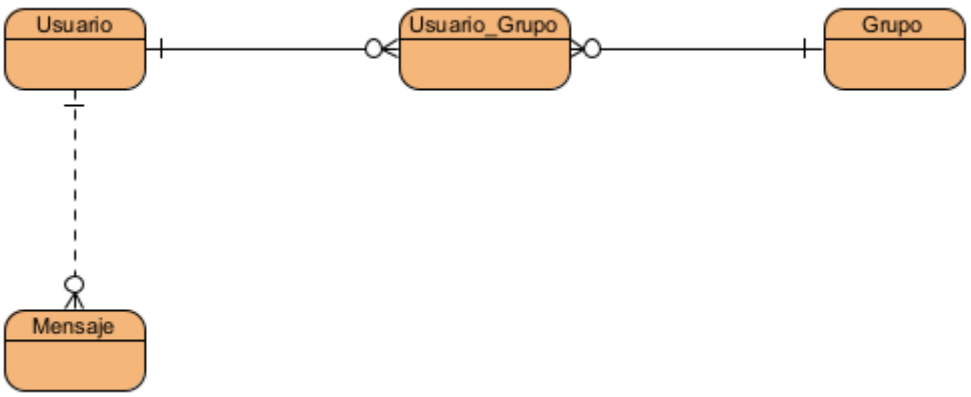


IMAGEN 21 DIAGRAMA DE CLASES

Una vez se hayan hecho uso de todos los fragmentos de código que se han ido describiendo en el documento, se puede obtener un resultado final como el que se puede observar en las imágenes de a continuación.

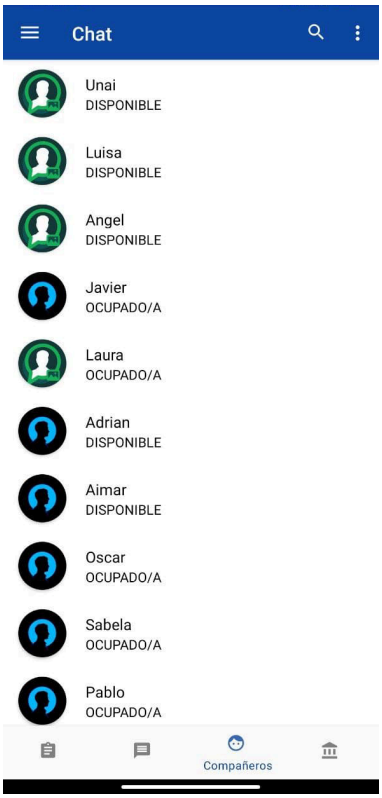


IMAGEN 22 LISTA DE COMPAÑEROS DE LA LIBRERÍA INCORPORADA EN UNA APLICACIÓN

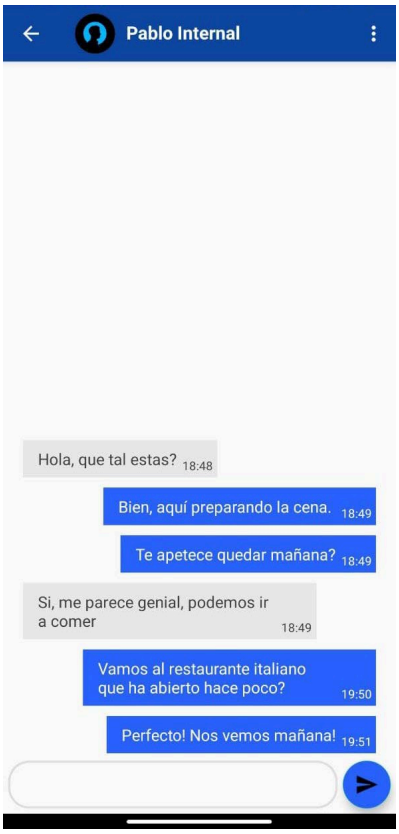


IMAGEN 23 CONVERSACIÓN DE LA LIBRERÍA DENTRO DE UNA APLICACIÓN

Protocolos de Uso

En este apartado, se van a detallar las instrucciones las cuales hay que seguir para poder incorporar correctamente la librería, tanto su parte de cliente, como el servidor. Con ello obtendremos un chat funcional dentro del entorno que se requiera.

Servidor

1. Requisitos previos:

    a. Tener una cuenta en Azure: es un requisito básico, ya que todo el servidor está basado en Microsoft, pero como se ha mencionado, esta solución es flexible, y se puede trasladar el proceso a otro proveedor si se desea.

    b. Crear la Base de Datos SQL Server: Se deberá crear un recurso de BD SQL en el panel de Azure.

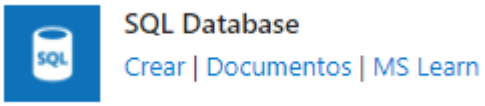


IMAGEN 24 RECURSO AZURE SQL

    c. Crear una AppService: es el servicio que se encarga de alojar las funcionalidades de la librería, es el núcleo del servidor.

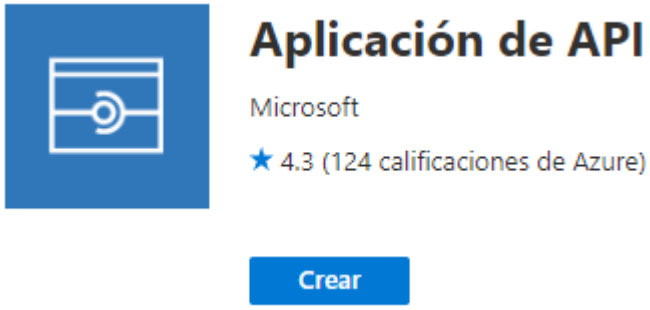


IMAGEN 25 RECURSO AZURE API

    d. Crear un Notification Hubs: con la cual posteriormente se conectará la API y se podrán recibir notificaciones push.

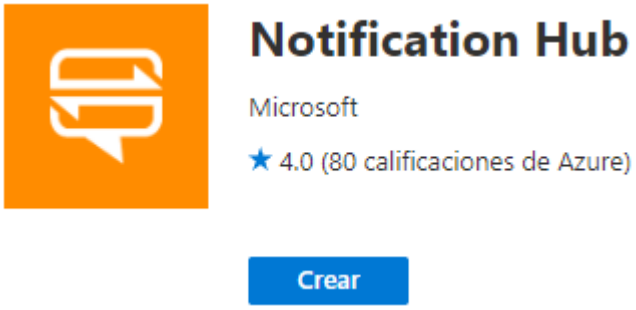


IMAGEN 26 RECURSO AZURE NOTIFICATION HUBS

2. Conectar Visual Studio a la BD creada:

Para ello abrimos Visual Studio, y mostramos el explorador de objetos.

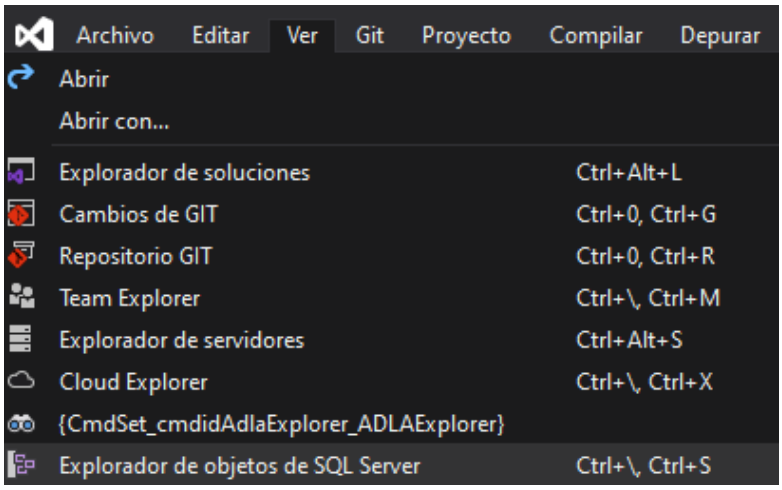


IMAGEN 27 EXPLORADOR ARCHIVOS VISUAL STUDIO

A continuación, iniciamos sesión con nuestras credenciales de Azure, lo que hará que nos aparezca las BBDD disponibles. Se deberá seleccionar la opción de Azure y seleccionar la BD correspondiente.

IMAGEN 28 CONECTAR BD A VISUAL STUDIO I

Finalmente, introducimos los datos de la BD que acabamos de crear en Azure.

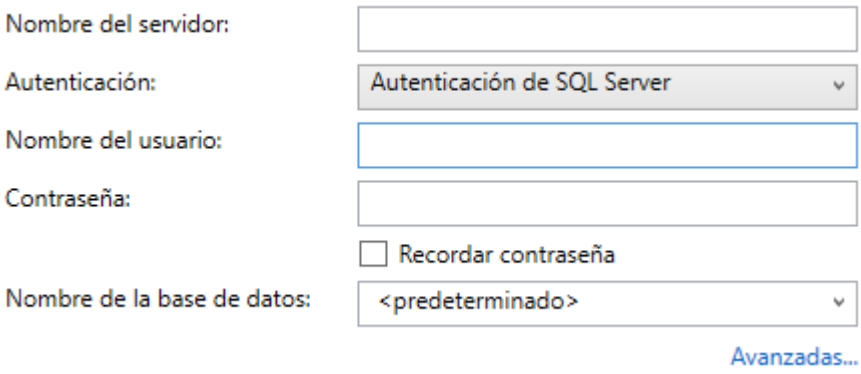


IMAGEN 29 CONECTAR BD A VISUAL STUDIO II

3. Con ayuda del script, crear las tablas

Se deberán generar las tablas para que el servicio funcione correctamente, para ello se ejecutaran los scripts que se proporcionan. Se trata de un archivo .sql que contiene una consulta y genera todas las tablas necesarias.

```
CREATE TABLE [dbo].[Message](
    [MessageId] [uniqueidentifier] NOT NULL,
    [FromUser] [uniqueidentifier] NOT NULL,
    [ToUSer] [uniqueidentifier] NULL,
    [Text] [varchar](max) NULL,
    [LastUpdateUTCLocal] [datetime] NOT NULL,
    [FileName] [varchar](300) NULL,
    [FileExtension] [varchar](10) NULL,
    [LastUpdateUTCInServer] [datetime] NULL,
    [GroupId] [uniqueidentifier] NULL,
 CONSTRAINT [PK_Message] PRIMARY KEY CLUSTERED
```

IMAGEN 30 CREACIÓN TABLA MENSAJE EN LA BD

Además, también crea las dependencias necesarias:

```
ALTER TABLE [dbo].[Message]  WITH NOCHECK ADD  CONSTRAINT [FK_Message_GroupId] FOREIGN KEY([GroupId])
REFERENCES [dbo].[ChatGroup] ([IdChatGroup])
```

IMAGEN 31 CREACIÓN DE LAS DEPENDENCIAS DE LA TABLA MENSAJE EN LA BD

4. Importar librería:

Se deberá crear un nuevo proyecto, e importar las siguientes librerías que estarán a disposición del usuario.

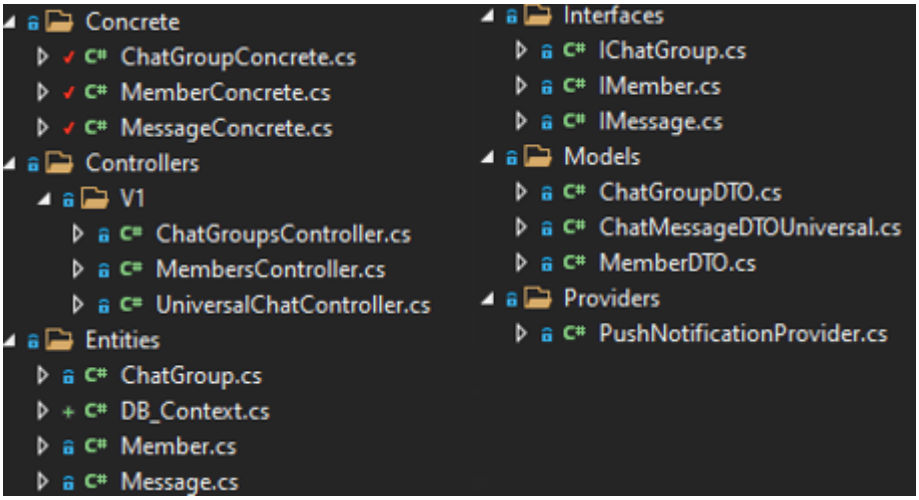


IMAGEN 32 CLASES A IMPORTAR EN EL PROYECTO DE VISUAL STUDIO

5. Configurar parámetros de conexión:

En la clase PushNotificationProvider, se deberá establecer los datos que correspondan con el NotificationHubs creado.

```
/// <summary>
/// Cadena de conexión para conectarse al Hub de notificaciones de desarrollo
/// </summary>
static private string hubTestConnectionString = "Endpoint=sb://
/// <summary>
/// Nombre del Hub de notificaciones de desarrollo
/// </summary>
static private string hubTestName = "
/// <summary>
/// Cadena de conexión para conectarse al Hub de notificaciones de producción
/// </summary>
static private string hubProductionConnectionString = "Endpoint=sb://
/// <summary>
/// Nombre del Hub de notificaciones de producción
/// </summary>
static private string hubProductionName = "
```

IMAGEN 33 PARÁMETROS A CAMBIAR EN LA CLASE PUSHNOTIFICATIONPROVIDER

Además, en los controllers, se podrá personalizar la ruta de las API's en caso de que se quiera.

```
[Route("api/V1/[controller]")]
```

IMAGEN 34 EJEMPLO DE VERSIÓN Y RUTA DE API

6. Publicar API:
   Finalmente, para tener la parte del servidor lista, se deberán publicar las API's con todas sus clases en el Portal de Azure, para ello, seleccionamos como destino Azure, introducimos los datos del servicio API que hemos creado anteriormente, lo seleccionamos y publicamos.

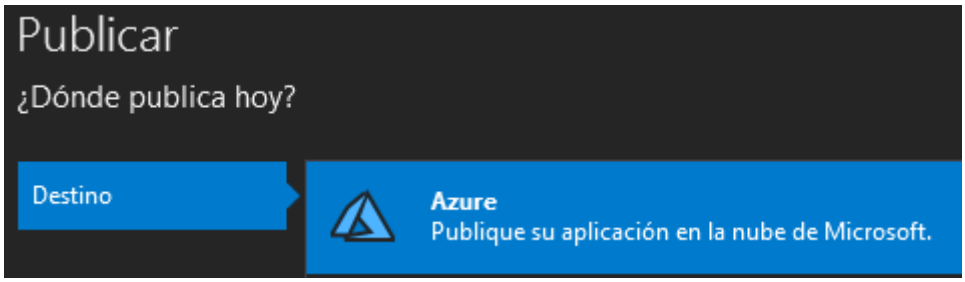


IMAGEN 35 PUBLICAR EN AZURE

Cliente

1. Crear/Abrir un proyecto en Android Studio:

Si no se cuenta con un proyecto de Android, se debe crear uno nuevo. Es importante a la hora de crear el proyecto, establecer como lenguaje Java.

2. Importar la librería:

Hay que seleccionar la opción de nuevo archivos y seleccionar importar un módulo

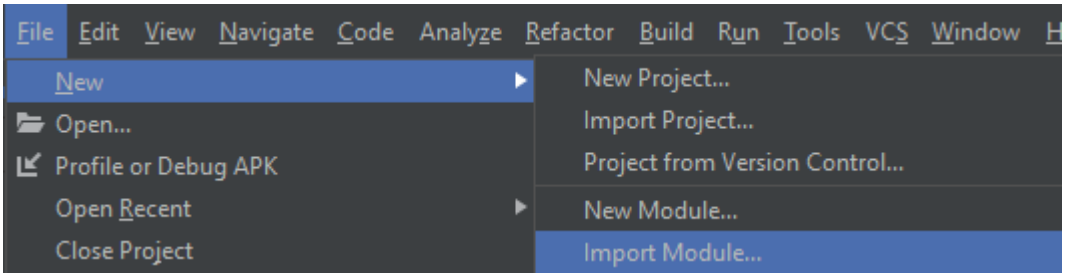


IMAGEN 36 ABRIR MENÚ DE IMPORTAR LIBRERÍA

En el cuadro que nos aparecerá, debemos seleccionar la ruta donde se encuentra el proyecto de la librería de chat. Posteriormente deberemos seleccionar el módulo de chat e importarlo.

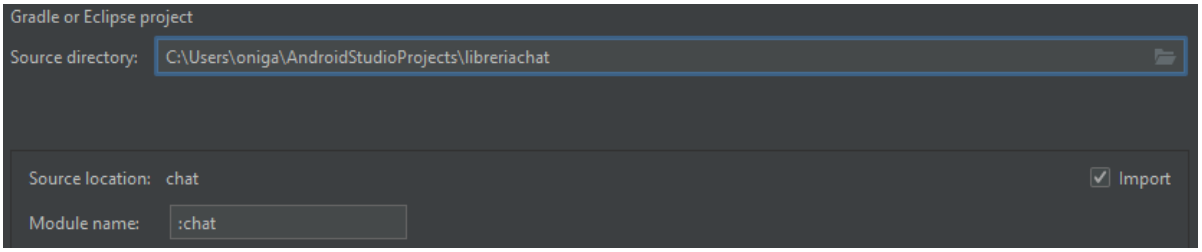


IMAGEN 37 IMPORTAR LIBRERÍA

3. Establecer dependencias:

Ahora debemos establecer las dependencias con la librería para poder empezar a utilizarla. Para ello vamos a abrir el gradle del proyecto y vamos a añadir la siguiente línea:

```
implementation project(':chat')
```

IMAGEN 38 AÑADIR DEPENDENCIAS EN GRADLE

Además, en las properties del gradle debemos asegurarnos de que están las siguientes líneas para evitar discrepancias en la compatibilidad.

```
android.enableJetifier = true
android.useAndroidX=true
```

IMAGEN 39 CONFIGURACIÓN GRADLE PROPERTIES

4. Configurar los parámetros de conexión

Para ello, como se tiene acceso a todo el código, iremos a la clase ServerConnection y definiremos nuestro servidor de API's y la versión que le hemos asignado.

```
public class ServerConnection extends AppCompatActivity {
    private final String API_URL = "https://[redacted].azurewebsites.net";
    private final String API_VERSION = "v1/";
```

IMAGEN 40 CONFIGURACIÓN URL SERVIDOR APIS

5. Configurar servicios de Google.

   Para poder utilizar la librería, se deberá crear una cuenta en Firebase, con la cual obtendremos acceso a los servicios de Google, pudiendo comunicar los clientes con el servidor. Deberemos crear un nuevo proyecto que deberá tener el mismo nombre que el proyecto en Android. Posteriormente se deberán seguir los pasos que nos indica la propia página para obtener e importar el archivo google-services.json en nuestro proyecto.

Ejemplo de Uso

Ahora que está ya todo configurado, podemos acceder a la librería como si estuviéramos en el mismo proyecto. Vamos a empezar a utilizar la librería, para ello vamos a crear un fragment del tipo ConversationsFragment, y reemplazaremos un fragment local, por el nuevo, el cual nos mostrará la misma pantalla que hemos explicado a lo largo del documento, y con ello la funcionalidad que conlleva.

Extra

Por último, como se ha mencionado antes, al incorporar dicha librería, tenemos acceso a todo su contenido, por lo cual podemos modificar todos los parámetros a nuestro gusto, modificar las vistas, los textos, todo. Además, si no queremos reemplazar el fragment en su totalidad, podemos crear una instancia de los objetos que nos interesan y usarlos a nuestro gusto.

Ejemplo real

Para poder ver que aspecto tiene la librería una vez insertada en una aplicación, se puede ir al **ANEXO 17** donde podemos observar la lista de contactos, al **ANEXO 18** donde vemos la función de cambiar nuestro estado a Disponible u Ocupado, en el **ANEXO 19** podemos ver que aspecto tiene una conversación individual y en el **ANEXO 20** se muestra la lista de grupos a los que pertenecemos. Finalmente, en el **ANEXO 21** se nos muestra la lista de conversaciones existentes.

Conclusiones y ampliaciones futuras

En este capítulo se van a exponer una serie de conclusiones sobre el trabajo que se ha llevado a cabo, el futuro de la herramienta y finalmente una valoración personal sobre el trabajo y lo que ha supuesto.

Conclusiones

Lo que se buscaba con la creación de la librería de chat, era facilitar la integración de esta en todos los sentidos, temporales, económicos y universal. Como se ha visto en el análisis de servicios existentes, no es la idea más innovadora ya que son varias plataformas las que ofrecen este servicio, pero la mayoría de ellas tienen algún inconveniente que hacen que no sean una opción real para un equipo de desarrollo o una empresa.

Algunas tienen un precio demasiado elevado, otras son muy estrictas en cuanto a personalización o en cuanto a la infraestructura con la que está construida. Y, por otro lado, muchas de ellas son directamente el servidor, es decir que la librería se conecta con sus servicios y después estos se conectan con el proveedor cloud correspondiente. Lo que hace que los datos personales sean compartidos con terceras personas, lo que es algo no deseable.

Así pues, intentando solventar todos estos inconvenientes se quiso crear nuestra librería. Una solución versátil, personalizable en todos los aspectos y fácilmente integrable. Además, esto se trata de una primera versión, si se quiere, a esta librería le queda mucha evolución por delante con muchas mejoras y nuevas funcionalidades.

Ampliaciones futuras

Este, como todo proyecto tecnológico, tiene mucho margen de mejora. A continuación, se van a explicar nuevas funcionalidades y cambios que podrían llevarse a cabo:

- Un servicio de intercambio de archivos, ya sean videos, audios, fotos o documentos. Y ya para cada aplicación final, decidiría cuales quiere permitir o si permite alguno.

- Envío de audios: Permitir a los usuarios comunicarse mediante notas de voz.

- Videollamadas: implementar un sistema de videollamadas para el chat, podría ser utilizado para conferencias en una aplicación interna de una empresa.

- Administrador de grupos: incluir herramientas para poder administrar los miembros de un grupo, así como los datos de este.

- iOS: ofrecer el mismo servicio para la plataforma móvil de Apple. Es decir, crear una librería para este sistema.

- Escritorio: ofrecer también una librería para clientes Web.

- Base de datos del cliente: Actualmente la base de datos que utilizada en el cliente viene encapsulada en la librería y a pesar de que es personalizable, en un futuro sería una buena opción modificarla para que sea integrable con la BD de la aplicación que incorpore la librería.

- Documentación: para facilitar el uso de la librería y la personalización de esta, en el futuro se debería crear una documentación detallada de todas las funcionalidades que otorga y cómo amoldarlas con la aplicación que la integra.

Valoración Personal

En cuanto a valoración personal, mis sensaciones son agridulces ya que no he podido cumplir con todos los objetivos que me había propuesto inicialmente. Pero por otro lado ha sido una buena experiencia porque he aprendido más sobre el funcionamiento de las API's, Azure y Android.

La idea inicial era presentar la librería para iOS también, pero por la gran curva de aprendizaje que me ha supuesto entender como realmente funciona Azure y las conexiones con Android, no me ha dado tiempo desarrollar una librería también para iOS.

Me ha servido también para entender que a pesar de que el pensamiento genérico que llevamos a cabo para plantear como vamos a implementar un programa, no siempre es válido porque hay muchas tecnologías que tienen su funcionamiento específico. Me ha pasado en este proyecto, que en ocasiones he querido diseñar según la forma lógica o común y no he conseguido ningún resultado a la hora de completar la implementación y darme cuenta después, de que era debido a algo bastante especifico de la tecnología.

Ha habido muchos momentos de frustración porque me he pasado semanas atascado con el mismo problema y no le podía dar solución, ese ha sido otro factor que ha hecho no poder completar mis objetivos.

Finalmente, y como opinión final, en general estoy contento con el desarrollo del proyecto, pero si es verdad que me quedo con ganas de más, porque ahora que tengo mejor controlado cómo funcionan las cosas, estoy seguro de que el avance sería mucho más rápido y conseguiría un resultado mejor. Lo ideal hubiese sido poder haber empezado el proyecto con un nivel de conocimiento similar, pero en un futuro estoy seguro de que me voy a volver a encontrar con situaciones de este tipo y que lo aprendido durante este proyecto me servirá para entonces.

Bibliografía

[Información sobre Pusher] Integrando Pusher con Android, URL: https://medium.com/@victor.garibayy/integrando-pusher-con-android-mensajer%C3%ADa-instant%C3%A1nea-18b479d7d900

[Pusher] Pusher, URL: https://pusher.com/

[Quickblox] Quickblox, URL: https://docs.quickblox.com/

[Stream] Stream, URL: https://getstream.io/tutorials/android-chat/

[Chat21] Chat21, URL: http://www.chat21.org/

[Sendbird] Sendbird, URL: https://sendbird.com/

[Metodologías de Desarrollo] ¿Qué tipos de metodologías de desarrollo de software existen? URL: https://www.becas-santander.com/es/blog/metodologias-desarrollo-software.html

[Visual Studio] Documentación, URL: https://docs.microsoft.com/es-es/visualstudio/windows/?view=vs-2019

[Android Studio] Documentación, URL: https://developer.android.com/docs

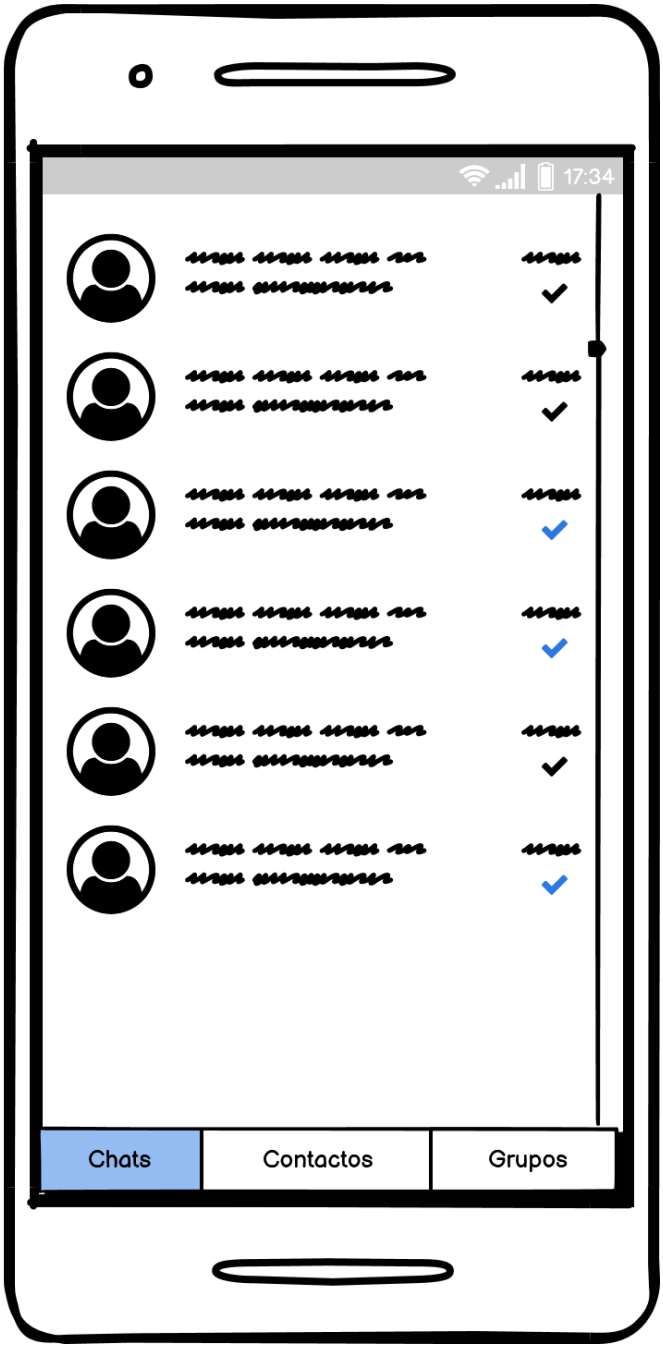
[Azure] Documentación, URL: https://docs.microsoft.com/es-es/azure/?product=featured

[SQL] Documentación, URL: https://docs.microsoft.com/es-es/sql/?view=sql-server-ver15

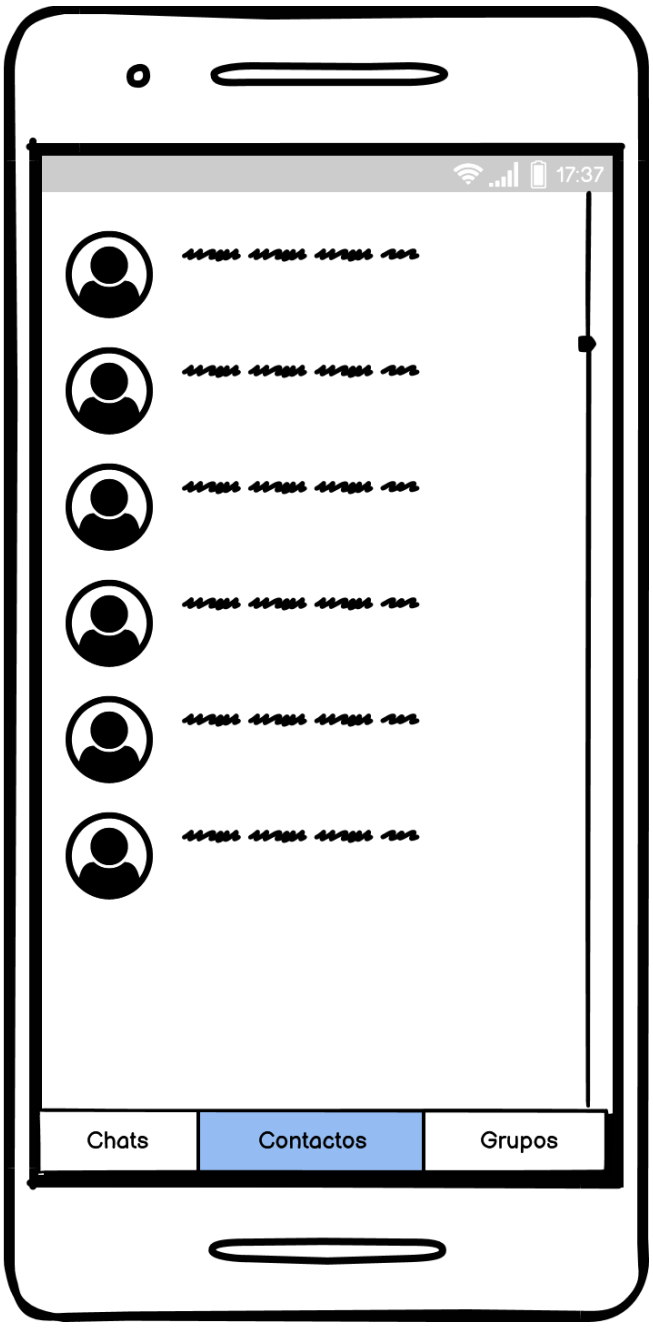
[Firebase] Servicios de Google, URL: https://firebase.google.com/docs/android/setup?hl=es-419#console

Anexos

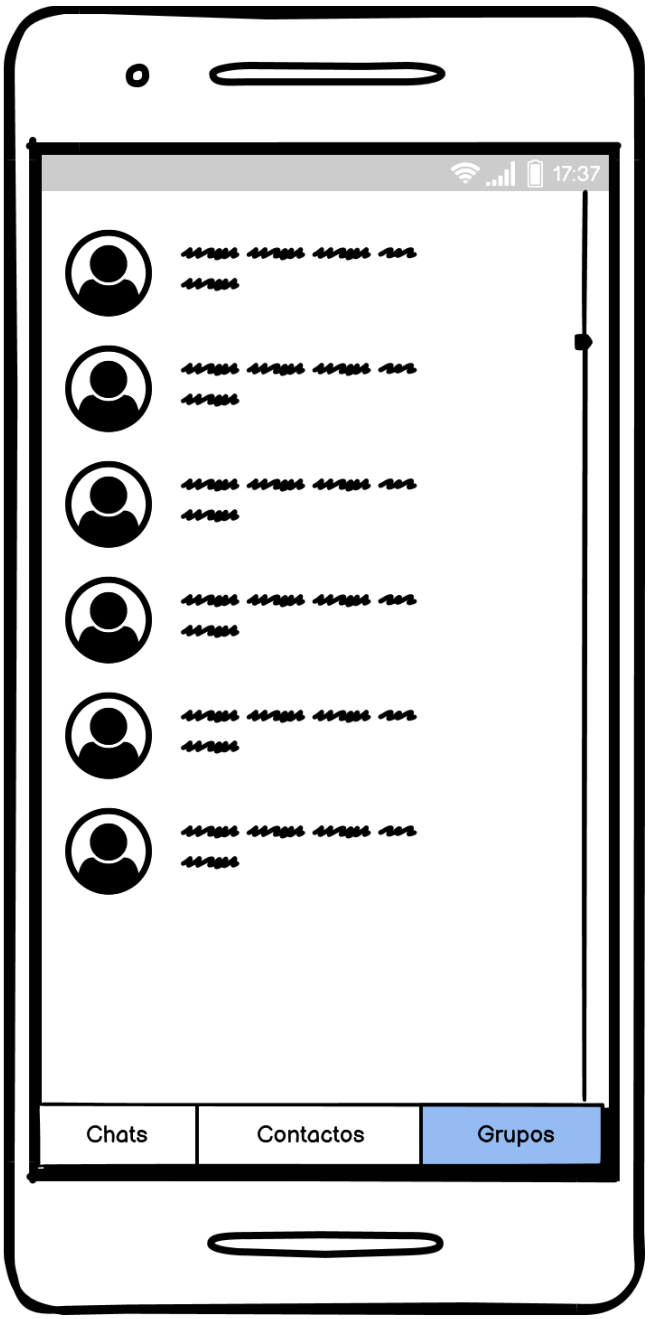
Anexo 1: Boceto de los diseños de pantalla



ANEXO 1 BOCETO DE LA PANTALLA DE CHATS



ANEXO 2 BOCETO DE LA PANTALLA DE CONTACTOS



ANEXO 3 BOCETO DE LA PANTALLA DE GRUPOS

Anexo 2: Fragmentos de código de la implementación

```
[HttpPost("SendChatMessage")]
0 referencias
public async Task<ActionResult> Post([FromBody] ChatMessageDTOUniversal chatMessage)
{
    System.Diagnostics.Trace.TraceInformation("Envío mensaje");
    var newChatMessage = mapper.Map<Message>(chatMessage);

    if (await message.SaveMessageInServerDatabase(newChatMessage))
    {
        var chatMessageDTOUniversal = mapper.Map<ChatMessageDTOUniversal>(newChatMessage);
        System.Diagnostics.Trace.TraceInformation("Llamada a envío push");
        PushNotificationProvider.SendChatMessage(chatMessageDTOUniversal);
        return new CreatedAtRouteResult(new { idMemberEnrolled = newChatMessage.MessageId }, chatMessageDTOUniversal);
    }
    else
        return BadRequest("Error al intetar almacenar el mensaje");
}
```

ANEXO 4 API ENVIAR MENSAJE

```
[HttpGet("GetAllNewMessages")]
0 referencias
public async Task<ActionResult> Get(Guid UserId, DateTime LastUpdateUTCInServer)
{
    var messagesResultDTO = new List<ChatMessageDTOUniversal>();
    var messages = await message.GetAllNewMessages(UserId, LastUpdateUTCInServer);

    if (message != null)
    {
        messages.ForEach(msg =>
        {
            messagesResultDTO.Add(mapper.Map<ChatMessageDTOUniversal>(msg));
        });

        return Ok(messagesResultDTO);
    }
    else
        return BadRequest("Error al obtener memsajes de chat");
}
```

ANEXO 5 API OBTENER MENSAJES

```
[HttpGet("GetAllGroupsByMemberEnrolled")]
0 referencias
public async Task<ActionResult<IEnumerable<ChatGroupDTO>>> GetAllGroupsByMemberEnrolled(Guid idMemberEnrolled)
{
    var chatGroups = await chatGroup.GetAllChatGroupsByMemberEnrolled(idMemberEnrolled);

    if (chatGroups == null)
        return NotFound();
    else
    {
        chatGroups.ForEach(group =>
        {
            if (!string.IsNullOrEmpty(group.Image))
            {
                group.Image = storage.GetImage(storageConfig.FileContainerName, group.Image).Result;
                group.MemberEnrolledInChatGroup = null;
            }
        });

        return Ok(mapper.Map<List<ChatGroupDTO>>(chatGroups));
    }
}
```

ANEXO 6 API OBTENER GRUPOS

```
[HttpPost("ChangeUserStatus")]
0 referencias
public async Task<ActionResult> Post(Guid idMemberEnrolled, String status)
{
    if (await _memberEnrolled.ChangeUserStatus(idMemberEnrolled, status))
    {
        return NoContent();
    }
    else
    {
        return BadRequest("Error al tratar de actualizar el estado");
    }

}
```

ANEXO 7 CAMBIAR ESTADO DEL USUARIO

```
public async Task<IEnumerable<Message>> GetAllNewMessages(Guid UserId, DateTime LastUpdateUTCInServer)
{
    //Se obtienen las Id's de todos los grupos en los cuales esta involucrado el usuario
    var groupsEnrolled = await context.MemberEnrolledInChatGroup
        .Where(g => g.IdMemberEnrolled.Equals(UserId))
        .Select(g => g.IdChatGroup)
        .ToListAsync();

    //Se obtienen todos los mensajes individuales del usuario
    var messagesResult = await context.Message
        .Include(msg => msg.MessageInfo)
        .Where(msg => msg.ToUser.Equals(UserId) || msg.FromUser.Equals(UserId) && msg.GroupId.Equals(null))
        .Where(msg => msg.LastUpdateUTCInServer > LastUpdateUTCInServer)
        .ToListAsync();

    //Se obtienen todos los mensajes de grupo y se añaden a messagesResult
    foreach (var grp in groupsEnrolled)
    {
        var messages = await context.Message
            .Include(msg => msg.MessageInfo)
            .Where(msg => msg.GroupId.Equals(grp))
            .Where(msg => msg.LastUpdateUTCInServer > LastUpdateUTCInServer)
            .ToListAsync();

        messagesResult.AddRange(messages);
    }

    return messagesResult;
}
```

ANEXO 8 CONCRETE OBTENER NUEVOS MENSAJES

```
public async Task<bool> SaveMessageInServerDatabase(Message chatMessage)
{
    // Get status code of Message
    var status = await context.MessageStatus.FirstAsync(sts => sts.StatusInfo.Equals("Enviado"));
    chatMessage.LastUpdateUTCInServer = DateTime.UtcNow;
    chatMessage.MessageInfo.Add(new MessageInfo
    {
        Date = DateTime.UtcNow,
        MessageFromUser = chatMessage.FromUser,
        MessageMessageId = chatMessage.MessageId,
        MessageStatusStatusCode = status.StatusCode
    });

    context.Add(chatMessage);
    return await context.SaveChangesAsync() > 0;
}
```

ANEXO 9 CONCRETE GUARDAR MENSAJE EN LA BD

```
public async Task<IEnumerable<ChatGroup>> GetAllChatGroupsByMemberEnrolled(Guid idMemberEnrolled)
{
    var resultGroup = new List<ChatGroup>();

    var result = await context.MemberEnrolledInOrganization
        .Include(icl => icl.MemberEnrolledInChatGroup)
            .ThenInclude(icl => icl.IdChatGroupNavigation)
        .Where(grp => grp.IdMemberEnrolled == idMemberEnrolled).FirstOrDefaultAsync();

    foreach (var memberGroup in result.MemberEnrolledInChatGroup)
    {
        resultGroup.Add(memberGroup.IdChatGroupNavigation);
    }

    return resultGroup;
}
```

ANEXO 10 CONCRETE OBTENER GRUPOS

```
public class ChatMessageDTOUniversal
{
    2 referencias
    public Guid MessageId { get; set; }
    2 referencias
    public Guid FromUser { get; set; }
    3 referencias
    public Guid? ToUser { get; set; }

    [MaxLength(1000, ErrorMessage = "The message exceeds the number of characters allowed")]
    2 referencias
    public string Text { get; set; }
    2 referencias
    public DateTime LastUpdateUTCLocal { get; set; }
    2 referencias
    public string FileName { get; set; }
    2 referencias
    public string FileExtension { get; set; }
    2 referencias
    public DateTime? LastUpdateUTCInServer { get; set; }
    2 referencias
    public Guid? GroupId { get; set; }
    2 referencias
    public virtual ICollection<MessageInfoDTO> MessageInfo { get; set; }
    0 referencias
    public virtual MemberEnrolledDTO FromUserNavigation { get; set; }
    0 referencias
    public virtual MemberEnrolledDTO ToUserNavigation { get; set; }
```

ANEXO 11 DTO ESTRUCTURA MENSAJE

```
public class ChatGroupDTO
{
    3 referencias
    public Guid IdChatGroup { get; set; }
    2 referencias
    public Guid IdMemberEnrolled { get; set; }
    4 referencias
    public int IdOrganization { get; set; }
    6 referencias
    public string GroupName { get; set; }
    5 referencias
    public string Image { get; set; }
    4 referencias
    public DateTime? CreateAt { get; set; }
    2 referencias
    public bool State { get; set; }

    2 referencias
    public virtual MemberEnrolledDTO MemberEnrolled { get; set; }
    2 referencias
    public virtual OrganizationDTO Organization { get; set; }
    2 referencias
    public virtual ICollection<MemberEnrolledInChatGroupDTO> MembersEnrolled { get; set; }
}
```

ANEXO 12 DTO ESTRUCTURA GRUPO

```
public class MemberEnrolledDTO
{
    3 referencias
    public Guid IdMemberEnrolled { get; set; }
    2 referencias
    public Guid? IdMember { get; set; }
    2 referencias
    public int? IdOrganization { get; set; }
    2 referencias
    public int? IdDepartment { get; set; }
    2 referencias
    public int? IdPosition { get; set; }
    2 referencias
    public int? IdMemberType { get; set; }
    2 referencias
    public string AccessId { get; set; }
    2 referencias
    public string Name { get; set; }
    2 referencias
    public string Lastname { get; set; }
    2 referencias
    public string Nickname { get; set; }
    2 referencias
    public string WorkPhone { get; set; }
    2 referencias
    public string WorkMail { get; set; }
    4 referencias
    public string Password { get; set; }
    6 referencias
    public string Image { get; set; }
    2 referencias
    public bool State { get; set; } = true;
```

ANEXO 13 DTO USUARIOS

```java
public void sendChatMessage(Message message, final FutureCallback<ApiResponse> callback){
    final ListenableFuture<ServiceFilterResponse> request =
            connectionServerApi.invokeApi( apiName: API_VERSION + API_POST_SEND_MESSAGE, getBodyData(message),
                    POST, getAuthorizationHeader(), getParameters(null));
    Futures.addCallback(request, new FutureCallback<ServiceFilterResponse>() {
        @Override
        public void onSuccess(ServiceFilterResponse result) {
            ApiResponse apiResponse = getObjectResponse(result, ApiResponse.class);
            callback.onSuccess(apiResponse);
        }

        @Override
        public void onFailure(Throwable t) { callback.onFailure(t); }
    }, Runnable::run);
}
```

ANEXO 14 MÉTODO DE ENVÍO DE MENSAJE DESDE CLIENTE

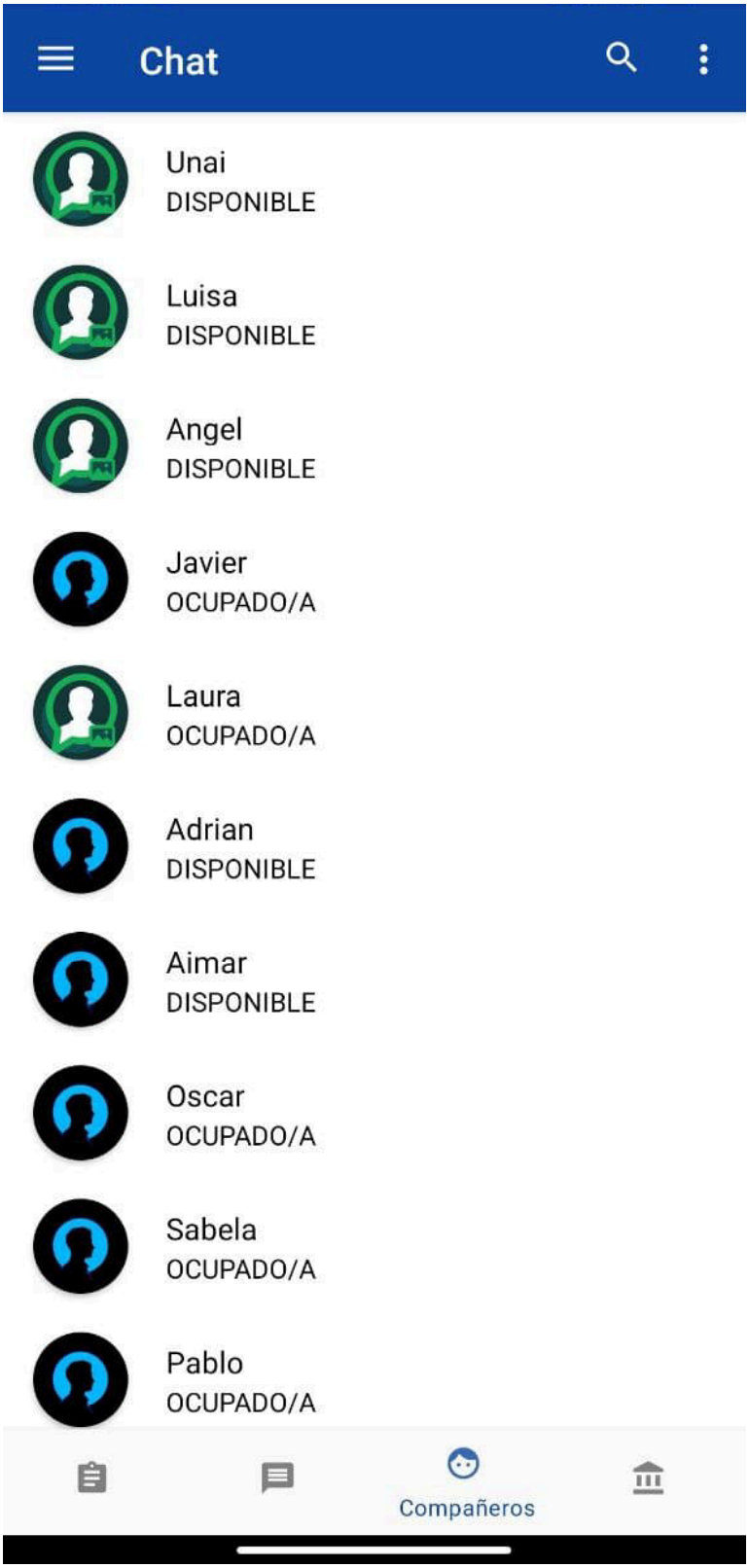
```java
public void getAllNewChatMessages(String idMemberEnrolled, String lastUpdate, final FutureCallback<Message[]> callback){
    Pair<String, String> pairUserId = new Pair<>(USER_ID, idMemberEnrolled);
    Pair<String, String> pairLastUpdate = new Pair<>(LAST_UPDATE_UTC_IN_SERVER, lastUpdate);
    java.util.List<Pair<String, String>> parametersCall = new ArrayList<>();
    parametersCall.add(pairUserId);
    parametersCall.add(pairLastUpdate);
    final ListenableFuture<ServiceFilterResponse> request =
            connectionServerApi.invokeApi( apiName: API_VERSION + API_GET_ALL_NEW_CHAT_INDIVIDUAL_MESSAGES,
             content: null, GET, getAuthorizationHeader(), getParameters(parametersCall));
    Futures.addCallback(request, new FutureCallback<ServiceFilterResponse>() {
        @Override
        public void onSuccess(ServiceFilterResponse result) {
            Message[] message = getObjectResponse(result, Message[].class);
            callback.onSuccess(message);
        }

        @Override
        public void onFailure(Throwable t) { callback.onFailure(t); }
    }, Runnable::run);
}
```

ANEXO 15 MÉTODO DE OBTENER MENSAJES DESDE CLIENTE

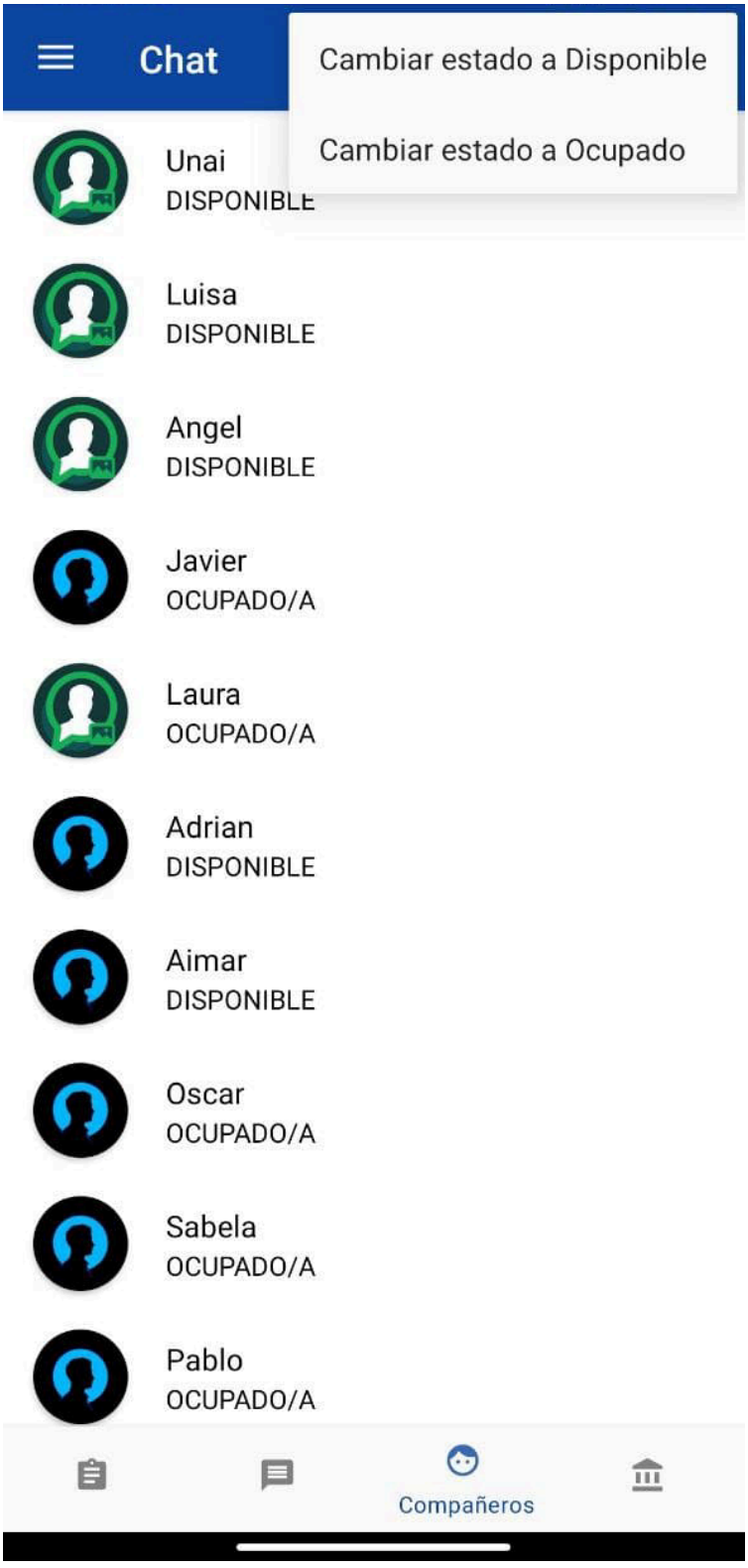
```java
public void changeUserStatus(String idMemberEnrolled, String status, final FutureCallback<ApiResponse> callback){
    Pair<String, String> pairIdMemberEnrolled = new Pair<>(ID_MEMBER_ENROLLED, idMemberEnrolled);
    Pair<String, String> pairStatus = new Pair<>(STATUS, status);
    java.util.List<Pair<String, String>> parametersCall = new ArrayList<>();
    parametersCall.add(pairIdMemberEnrolled);
    parametersCall.add(pairStatus);
    final ListenableFuture<ServiceFilterResponse> request = connectionServerApi.invokeApi( apiName: API_VERSION + API_CHANGE_STATUS,
             content: null, POST, getAuthorizationHeader(), getParameters(parametersCall));
    Futures.addCallback(request, new FutureCallback<ServiceFilterResponse>() {
        @Override
        public void onSuccess(ServiceFilterResponse result) {
            ApiResponse apiResponse = getObjectResponse(result, ApiResponse.class);
            callback.onSuccess(apiResponse);
            System.out.println(result);
        }

        @Override
        public void onFailure(Throwable t) {
            callback.onFailure(t);
        }
    }, Runnable::run);
}
```

ANEXO 16 MÉTODO DE CAMBIAR ESTADO DESDE EL CLIENTE

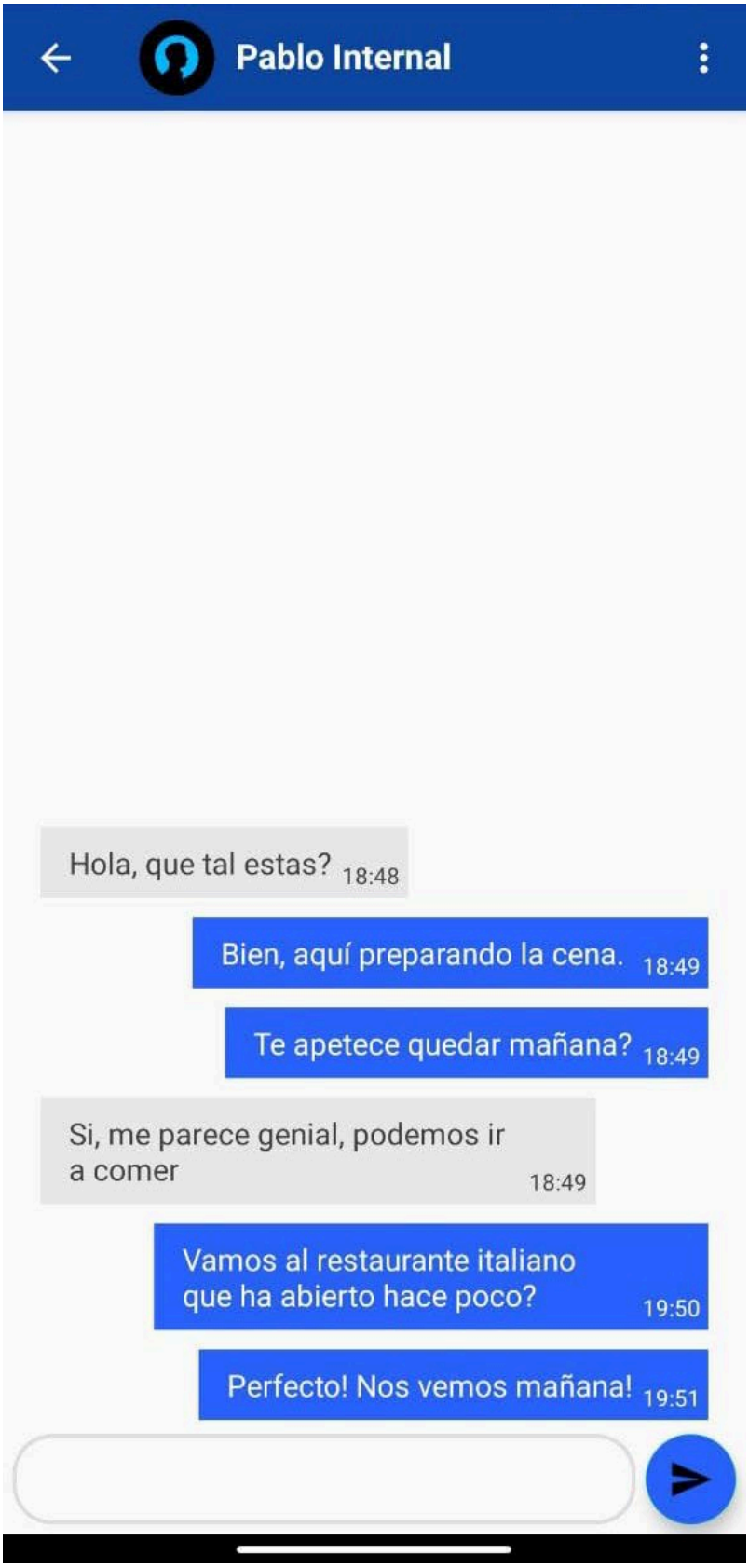
Anexo 3: Capturas de pantalla de la librería dentro de una aplicación



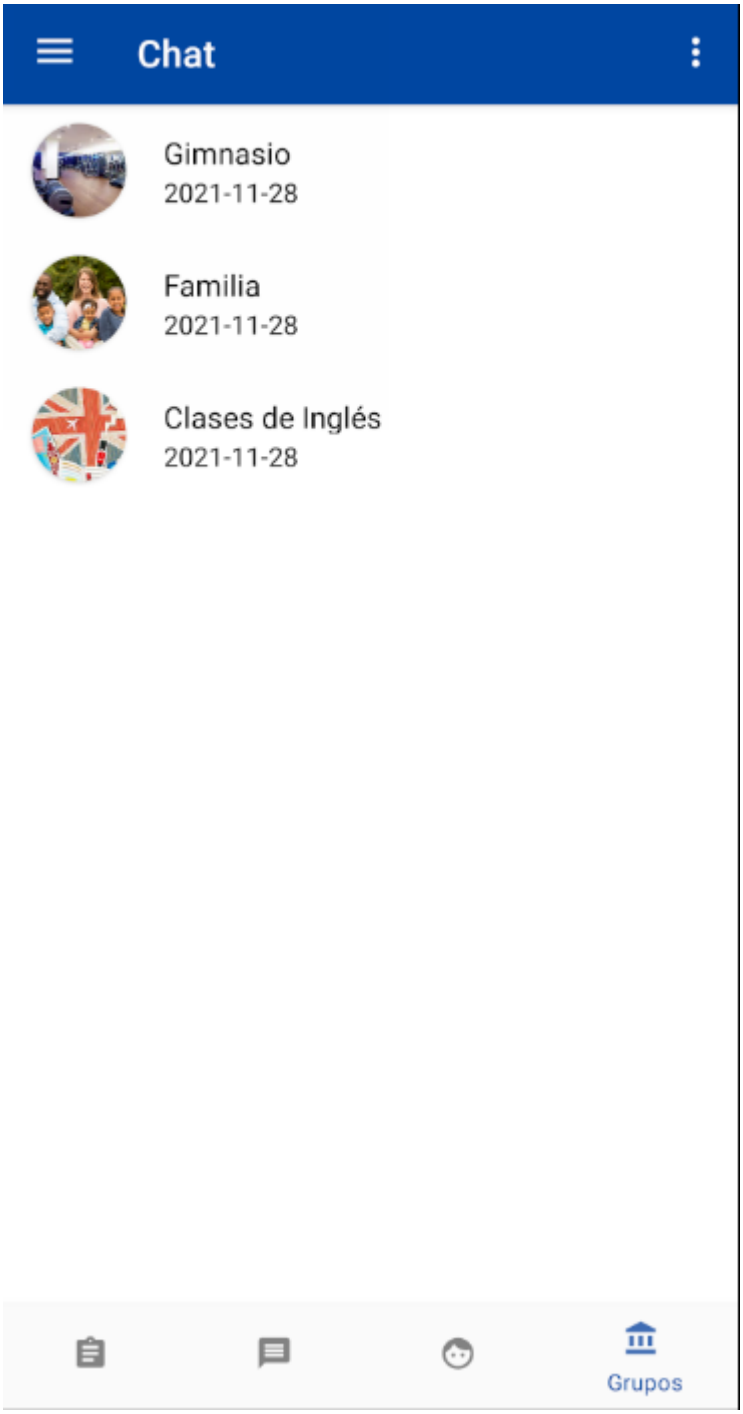
ANEXO 17 CAPTURA DE PANTALLA DE LA LISTA DE CONTACTOS DE LA LIBRERÍA DENTRO DE UNA APLICACIÓN



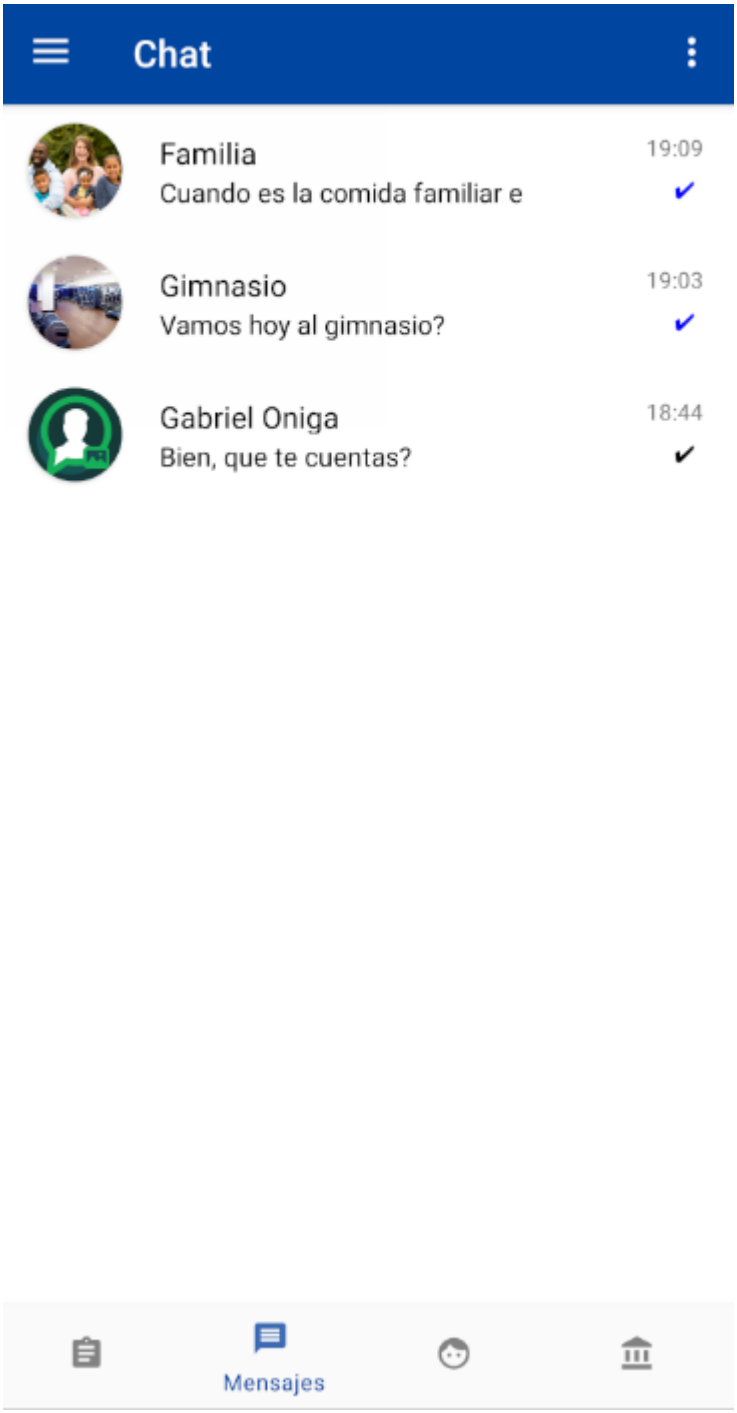
ANEXO 18 CAPTURA DE PANTALLA DE LA OPCIÓN DE CAMBIAR DE ESTADO DE LA LIBRERÍA EN UNA APLICACIÓN



ANEXO 19 CAPTURA DE PANTALLA DE UNA CONVERSACIÓN DE LA LIBRERÍA DENTRO DE UNA APLICACIÓN



ANEXO 20 CAPTURA DE PANTALLA DE LA LISTA DE GRUPOS DE LA LIBRERÍA DENTRO DE UNA APLICACIÓN



ANEXO 21 CAPTURA DE PANTALLA DE LA LISTA DE CONVERSACIONES DE LA LIBRERÍA EN UNA APLICACIÓN