Bluetooth App

Bluetooth App commits

Finalized Offline Bluetooth Chat App: Extreme Detail

This application is conceived as a **robust, secure, and user-friendly communication solution designed for environments where traditional network access (internet, cellular) is unavailable or unreliable.** Its core purpose is to enable **direct, peer-to-peer (P2P) communication between two Android devices using only Bluetooth Classic (RFCOMM)**, while ensuring **end-to-end encryption (E2EE)** for all transmitted data.

1. Core Purpose & Vision

The app provides a lifeline for communication in critical scenarios like disaster zones, remote outdoor activities, crowded events with network congestion, or simply when users prefer complete data privacy and local-only connectivity. It acts as an independent, self-contained communication bubble between two individuals.

2. Connectivity Model

The app leverages Bluetooth Classic's **RFCOMM (Radio Frequency Communication)** protocol, which emulates a serial port. This allows for stable, continuous data streams necessary for chat and file transfers.

- **Client-Server for Connection Establishment:**
    - One device acts as the **"Server"**: It initiates a listening state, making itself discoverable and waiting for an incoming connection request on a predefined, universally unique identifier (UUID) specific to your app.
    - The other device acts as the **"Client"**: It actively scans for nearby Bluetooth devices (including the server) and, upon finding the target device, attempts to initiate a connection to that device's specific UUID.
    - Once the client's connection attempt is accepted by the server, a **bidirectional** `BluetoothSocket` is established.

- **Peer-to-Peer for Data Exchange:**
    - Crucially, once the `BluetoothSocket` is established, the connection becomes symmetrical. Both devices are now "peers" and can simultaneously send and receive data over the same socket. There's no longer a strict server-client hierarchy for communication; data flows in both directions.
- **Persistent Connections:** The app utilizes an Android `ForegroundService` to maintain the active Bluetooth connection and communication threads in the background. This ensures that the chat continues, and messages/files can be sent/received, even if the user navigates away from the app's UI or the screen locks. A persistent notification in the status bar indicates the service's active status.

3. End-to-End Encryption (E2EE) & Authentication

This is a cornerstone of the app's security and a major differentiating factor. All data exchanged between the two devices is encrypted such that only the two communicating devices can read it.

- **Post-Connection E2EE Handshake:** The E2EE setup occurs immediately after the raw Bluetooth socket is established, *before* any user-generated data is sent.
- **Diffie-Hellman Key Exchange (DHKE):**
    - **Purpose:** This is the foundational cryptographic primitive for establishing a **shared secret key** over an insecure channel. It solves the problem of "how do two people agree on a secret code when a spy might be listening to everything they say?"
    - **Process:** Both devices generate a temporary (ephemeral) pair of cryptographic keys: a private key (kept secret) and a public key (shared openly). They exchange their public keys over the established Bluetooth socket. Through a clever mathematical process involving their own private key and the *other device's public key*, both devices independently compute the *exact same* shared symmetric secret. This shared secret is *never* transmitted directly.
    - **Result:** A unique, high-entropy symmetric key for the current communication session.

- **AES-256 GCM for Data Encryption:**
  - **Algorithm:** The Advanced Encryption Standard (AES) operating in Galois/Counter Mode (GCM) with a 256-bit key length.
  - **Purpose:** This is the workhorse symmetric encryption algorithm that uses the shared secret key derived from DHKE to **encrypt and decrypt all actual application data** (messages, files, location).
  - **Process:**
    - **Encryption:** When data is sent, the app takes the plaintext (e.g., your message), the DHKE-derived session key, and a unique **Initialization Vector (IV)**. It then scrambles the plaintext into ciphertext.
    - **Decryption:** The receiving app uses the same DHKE-derived session key and the transmitted IV to unscramble the ciphertext back into readable plaintext.
  - **IV Generation & Management:** Each new encryption operation uses a unique IV. The IV is typically sent alongside the encrypted data (it doesn't need to be secret, only unique for each operation with the same key). GCM also provides data authenticity, meaning the receiver can detect if the encrypted data has been tampered with in transit.
- **Authentication & Man-in-the-Middle (MitM) Attack Protection:**
  - **The MitM Problem:** DHKE alone is vulnerable to a "Man-in-the-Middle" attack. An attacker could intercept the public key exchange, swap their own public keys, and establish two separate secure connections (one with each legitimate device). The attacker then decrypts all traffic from one side, re-encrypts it with the other key, and forwards it, making both legitimate users believe they are talking directly to each other.
  - **Solution: Trust on First Use (TOFU) Model:**
    - **Initial Trust:** When two devices connect for the *very first time*, after the DHKE, the app automatically stores a cryptographic "fingerprint" (a unique hash derived from the peer's public key) associated with that Bluetooth device's identifier (MAC address). This initial trust is established assuming no MitM attacker is present at that exact moment.

        - **Subsequent Connections:** For all future connections with that same peer, the app automatically verifies that the peer is presenting the *same* public key fingerprint as the one previously stored. If it matches, the connection is deemed secure and verified.
        - **Critical Security Alert for Key Changes:** If a previously "trusted" device's fingerprint *changes*upon reconnection, the app immediately displays a **full-screen, blocking, red-alert warning dialog**. This alerts the user to a potential MitM attack or key compromise. The user is given options to "Disconnect" (recommended) or "Proceed Anyway (Unsafe)". This is crucial as it protects against *future* MitM attacks after the first connection.
    - **Optional Interactive Short Authentication String (SAS) Verification:**
        - **Purpose:** For users seeking the highest assurance, this feature allows them to manually verify the identity of their peer, mitigating the initial MitM vulnerability of TOFU.
        - **Process:** After the DHKE, both devices compute a short, human-readable Short Authentication String (SAS) derived from the shared secret. Instead of long, complex hexadecimal strings, the app converts this SAS into a sequence of **easy-to-read words** (e.g., "FISH MOUNTAIN CLOCK").
        - **User Interaction:** Users navigate to a dedicated "Security Verification" screen. They are instructed to verbally compare the word codes (e.g., "What's the code on your screen? Mine is FISH MOUNTAIN CLOCK.")
        - **Interactive UI:** The app provides an interactive input field where the user can type or select the words they see on *their friend's device*. As they input each word, the app provides **real-time visual feedback (e.g., green for a match, red for a mismatch)**, guiding them to accurately complete the verification. If all words match, the connection is marked as definitively "verified."

4. Communication & Content Sharing

The app supports a variety of data types, handled robustly over the Bluetooth stream.

- **Text Chat:** Standard instant messaging, displaying messages in distinct "my message" and "their message" bubbles.
- **Multimedia (Images & Videos):**
  - Users can select images/videos from their device's gallery or camera.
  - Images/videos are optionally resized/compressed to optimize transfer speed over Bluetooth.
  - Sent content is encrypted and streamed.
  - Received images/videos are displayed as **thumbnails** directly within the chat interface. Tapping a thumbnail opens a dedicated **viewer** (full-screen image viewer, basic video player).
- **Generic File Sharing (e.g., PDFs, documents):**
  - Users can select any file type from their device's storage.
  - Files are chunked, encrypted, and streamed securely.
  - The chat UI displays the **file name, file size, and transfer progress**.
  - Upon successful receipt, the user can choose to "Open" the file (which uses the Android system to launch a suitable external app, e.g., a PDF viewer for a PDF) or "Save to Device."
- **Message Framing Protocol:** To enable sending different data types over a single byte stream, the app implements a simple custom protocol. Each piece of data (text, image, file, location) is prefixed with a **type header** (indicating what kind of data it is) and a **length header** (indicating the size of the encrypted payload). This allows the receiving device to correctly parse, decrypt, and display the incoming information.

5. Offline Location Sharing

This feature provides relative location information without relying on online maps.

- **GPS Data Acquisition:** The app uses the device's built-in GPS receiver (`LocationManager`) to obtain the current latitude, longitude, and accuracy of the user's position.

- **Encrypted Transmission:** These coordinates are encrypted using the E2EE session key and sent as a special message type.
- **In-App Distance Calculation:** Upon receiving the other user's location, the app calculates the **straight-line distance** between the two devices (using standard geospatial algorithms like the Haversine formula) and displays it directly in the chat (e.g., "They are 50 meters away").
- **Compass-like Directional Indicator:**
    - The app accesses the device's **magnetometer (compass) and accelerometer sensors**.
    - These sensor readings are used to determine the device's current orientation (which way it's facing in the real world).
    - The app then calculates the **bearing** (the compass direction from the local device to the received remote location).
    - A **simple arrow icon** is rendered within the location message bubble on the chat screen. This arrow **dynamically points towards the other user's location**, updating as the local device is physically rotated. This provides an intuitive, real-time "where are they?" visual cue without needing any maps.

6. User Experience (UI/UX) Principles

The app's design prioritizes clarity, intuitive flow, and reliable feedback, especially crucial for an offline, connection-dependent application.

- **Clear State Communication:** The UI constantly informs the user about the Bluetooth adapter status (on/off), connection status (scanning, connecting, connected, disconnected), and file transfer progress.
- **Intuitive Navigation:** A clean bottom navigation bar provides quick access to core functions: **"Connect/Devices"** (the default landing screen), **"Chats/Messages"**, and **"Security/Settings"**.
- **Minimalist & Clean Design:** Focuses on essential chat and connection features, avoiding clutter.

- **Actionable Error Messages:** Provides clear, user-friendly messages for connection failures, permission issues, or security alerts, guiding the user on how to resolve them.

7. Underlying Architectural Concepts

- **Modular Design:** The app is structured with clear separation of concerns (e.g., a dedicated Bluetooth service, a security/crypto module, UI components).
- **Reactive UI:** Utilizes Jetpack Compose for the UI, enabling a declarative and reactive user interface that efficiently updates in response to changes in connection status, incoming messages, and sensor data.
- **Persistent Data:** Employs a **Room database** to store chat message history and conversation metadata locally on the device, ensuring messages persist across app sessions. Trusted device fingerprints are also securely stored.

This application is not just a chat client; it's a demonstration of robust offline communication engineering, advanced cryptography, and thoughtful user experience design in constrained environments.

The "Message Framing Protocol" you described is absolutely essential for your application. It addresses a fundamental challenge of network communication, especially when dealing with various types of data over a continuous stream.

Here's the breakdown of why it's needed:

The Problem: Sockets are Byte Streams, Not Message Streams

When you establish a `BluetoothSocket` (or any typical network socket like TCP), you're essentially getting a **continuous stream of raw bytes**. The operating system doesn't inherently know where one "message" ends and another begins.

Imagine pouring water from a tap into a long pipe. If you pour a glass of water, then a bucket of water, then another glass, on the other end of the pipe, all you see is a continuous flow of water. You can't tell where one container's worth of water stopped and the next began unless you had some way of marking those boundaries at the pouring end.

Similarly, if you send:

1. "Hello" (5 bytes)
2. An image (20000 bytes)
3. "How are you?" (11 bytes)

The receiving device just gets a continuous stream of `5 + 20000 + 11 = 20016` bytes. It has no built-in mechanism to distinguish the boundaries of these distinct pieces of information, nor what *type* of information each segment represents.

The Solution: Message Framing

Your custom "Message Framing Protocol" provides the necessary structure and metadata to transform that raw, unstructured byte stream into discrete, meaningful "messages" that the application can understand and process.

Here's the **need** for each component:

1. **Type Header (Indicating What Kind of Data it Is):**
    - **Need:** Without a type header, the receiving device wouldn't know if the incoming bytes represent text, an image, a video, a file, or location data.
    - **Function:** It's like putting a label on a package saying "This package contains a book" or "This package contains groceries." This allows the receiving application to direct the bytes to the correct processing logic (e.g., "If it's text, display it in the chat bubble; if it's an image, pass it to the image decoder").
2. **Length Header (Indicating the Size of the Encrypted Payload):**
    - **Need:** This is crucial for defining the **message boundary**. Since the receiver gets a continuous stream, it needs to know *exactly how many bytes* belong to the current message before it can look for the next one.
    - **Function:** It's like putting the "weight" or "dimensions" on a package. The receiver reads this fixed-size length field first. Once it knows the expected length (e.g., "the next 500 bytes are part of this message"), it can then read precisely that many bytes from the stream, process them, and then know that the very next byte in the stream marks the beginning of the *next* message's header.

- **Without it:** The receiver would have to guess when a message ends, potentially reading too little (truncating a message) or too much (mixing parts of the next message). This would lead to corrupted data, broken messages, and the entire communication becoming unsynchronized.

In Summary:

Your Message Framing Protocol is vital because it:

- **Defines Message Boundaries:** Allows the receiver to know exactly where one message ends and the next begins within the continuous stream of bytes.
- **Enables Data Type Recognition:** Allows the receiver to understand what kind of data it has received (text, image, file, location) and process it appropriately.
- **Ensures Robustness:** Makes the communication resilient to varying message sizes and prevents data corruption by misinterpreting stream content.
- **Facilitates Decryption:** The length header specifically refers to the *encrypted payload's* size. This means the receiver knows exactly how many encrypted bytes to read before attempting decryption.

Essentially, it's the agreed-upon "language" or "format" that both your sending and receiving apps use to understand each other's transmissions over the raw Bluetooth pipe.

You can learn more about message framing in general computer networking concepts, which are often applied to stream-based protocols like TCP. What is Message Framing in Computer Networks?

Yes, absolutely! You can implement all of these concepts in Kotlin for Android. Android's SDK provides the necessary APIs, and Kotlin is the preferred language for Android development, offering a more concise and expressive way to write your code compared to Java.

Here's a conceptual overview of how you'd implement each part in Kotlin on Android:

1. Bluetooth Connection Management

- **Permissions:** You'll use the Android permission system to request `BLUETOOTH_SCAN`, `BLUETOOTH_ADVERTISE`, `BLUETOOTH_CONNECT`, and `ACCESS_FINE_LOCATION` (for discovery) at runtime using the `ActivityResultLauncher` API.
- **`BluetoothAdapter`:** Get the default Bluetooth adapter using `BluetoothAdapter.getDefaultAdapter()`.
- **Discoverability:** Use an `Intent` with `BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE` to make your device visible to others.
- **Scanning/Discovery:** Call `bluetoothAdapter.startDiscovery()` to find nearby devices. You'll need a `BroadcastReceiver` to listen for `BluetoothDevice.ACTION_FOUND` and `BluetoothAdapter.ACTION_DISCOVERY_FINISHED` events.
- **Server (Accepting Connections):**
  - Create a `BluetoothServerSocket` using `bluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, UUID)`.
  - In a background thread (e.g., a Kotlin Coroutine with `Dispatchers.IO`), call `serverSocket.accept()`. This is a blocking call that returns a `BluetoothSocket` when a connection is established.
- **Client (Initiating Connections):**
  - Get a `BluetoothDevice` object for the target device (e.g., from a discovery scan result or paired devices).
  - Create a `BluetoothSocket` using `device.createRfcommSocketToServiceRecord(UUID)`.
  - In a background thread, call `socket.connect()`. This is also a blocking call.
- **Communication (`ConnectedThread`):**
  - Once a `BluetoothSocket` is connected, obtain its `InputStream` and `OutputStream` using `socket.inputStream` and `socket.outputStream`.
  - Set up a `read` loop in a background coroutine to continuously read bytes from the `InputStream`.
  - Implement a `write` function to take byte arrays and write them to the `OutputStream`.

- Crucially, remember to close sockets and streams properly when done to release resources.

2. End-to-End Encryption (E2EE)

Android's Java Cryptography Extension (JCE) APIs are available in Kotlin.

- **Diffie-Hellman Key Exchange (DHKE):**
  - You'll use classes from `javax.crypto` and `java.security`.
  - **Key Pair Generation:** `KeyPairGenerator.getInstance("DH")` to generate your ephemeral DH `KeyPair` (private and public keys).
  - **Public Key Exchange:** Transmit your `PublicKey` (as `ByteArray` using `publicKey.encoded`) over the *raw* Bluetooth socket stream to the other device.
  - **Key Agreement:** On both sides, once you have your `PrivateKey` and the *other device's* *`PublicKey`*, you'll use `KeyAgreement.getInstance("DH")` and then `keyAgreement.init(yourPrivateKey)` followed by `keyAgreement.doPhase(otherPublicKey, true)`. Finally, `keyAgreement.generateSecret()` will yield the shared symmetric secret key (as a `ByteArray`).
  - **Key Derivation Function (KDF):** It's good practice to use a KDF (like PBKDF2 or HKDF) on the raw shared secret from DHKE to derive your final AES key. This ensures the key is exactly the right size and has good properties.
- **AES-256 GCM Encryption/Decryption:**
  - **Cipher Instance:** `Cipher.getInstance("AES/GCM/NoPadding")`. "NoPadding" is used with GCM because GCM handles padding intrinsically.
  - **Key Generation (for AES):** You'll derive your `SecretKey` (e.g., `SecretKeySpec` from the DH-derived shared secret).
  - **Initialization Vector (IV) Generation:** Use `SecureRandom` to generate a **unique, random 12-byte IV for each encryption operation**. This IV is *not* secret but must be unique.
  - **Encryption:** Initialize the `Cipher` in `ENCRYPT_MODE` with your `SecretKey` and an `IvParameterSpec` created from your IV. Call `cipher.doFinal(plaintextByteArray)`. This

will return the ciphertext, which includes the Authentication Tag from GCM. You'll send the IV *concatenated* with the ciphertext.

- **Decryption:** Read the IV from the received bytes, then initialize the `Cipher` in `DECRYPT_MODE` with the same `SecretKey` and `IvParameterSpec`. Call `cipher.doFinal(receivedCiphertextWithTagByteArray)`. GCM will automatically verify the tag during decryption; if it fails (`AEADBadTagException`), the data was tampered with.

3. Message Framing Protocol

You'll implement this manually as part of your `ConnectedThread`'s read/write logic.

- **Data Structure:** Define an `enum class MessageType { TEXT, IMAGE, FILE, LOCATION, DH_PUBLIC_KEY, SAS_VERIFICATION }`.
- **Serialization (Sending):**
  - Convert your `MessageType` to a byte (e.g., `messageType.ordinal.toByte()`).
  - Convert the *length of your encrypted payload* (e.g., `Int` representing bytes) into a fixed-size byte array (e.g., 4 bytes).
  - Concatenate these: `[TYPE_BYTE] + [LENGTH_4_BYTES] + [ENCRYPTED_PAYLOAD_BYTES]`.
  - Write this complete byte array to the `OutputStream`.
- **Deserialization (Receiving):**
  - In your `read` loop:
    - First, read a fixed number of bytes for the **Type Header** (e.g., 1 byte).
    - Then, read a fixed number of bytes for the **Length Header** (e.g., 4 bytes) and convert it back to an `Int`.
    - Finally, read *exactly* that many bytes (`length`) for the **Encrypted Payload**.
  - Once you have the full encrypted payload and its type, pass it to your decryption and processing logic.

4. Multimedia & Location

- **Image/File Selection:** Use Android's `ActivityResultContracts.PickVisualMedia` for images/videos or `Intent.ACTION_GET_CONTENT` for general files.
- **Image Resizing/Compression:** Libraries like Coil, Glide, or even Android's own `BitmapFactory` can help.
- **File Streaming:** For large files, you'll need to read them in chunks from `FileInputStream` and write them in chunks to the `OutputStream` within your framing protocol. Include chunking info (e.g., total chunks, current chunk index) in your header or payload.
- **Location:** Use `LocationManager` and request `ACCESS_FINE_LOCATION` permission. Implement a `LocationListener` to get periodic updates. Send `latitude`, `longitude`, and `accuracy` as part of your `LOCATION` message payload.
- **Distance/Direction:**
  - `android.location.Location.distanceBetween()` for distance calculation.
  - `SensorManager` (for `TYPE_ACCELEROMETER` and `TYPE_MAGNETIC_FIELD`) to get raw sensor data. You'll then use `SensorManager.getRotationMatrix()` and `SensorManager.getOrientation()` to calculate the device's orientation relative to magnetic north, which you can then combine with the bearing to the target location.

5. Persistence (Room Database)

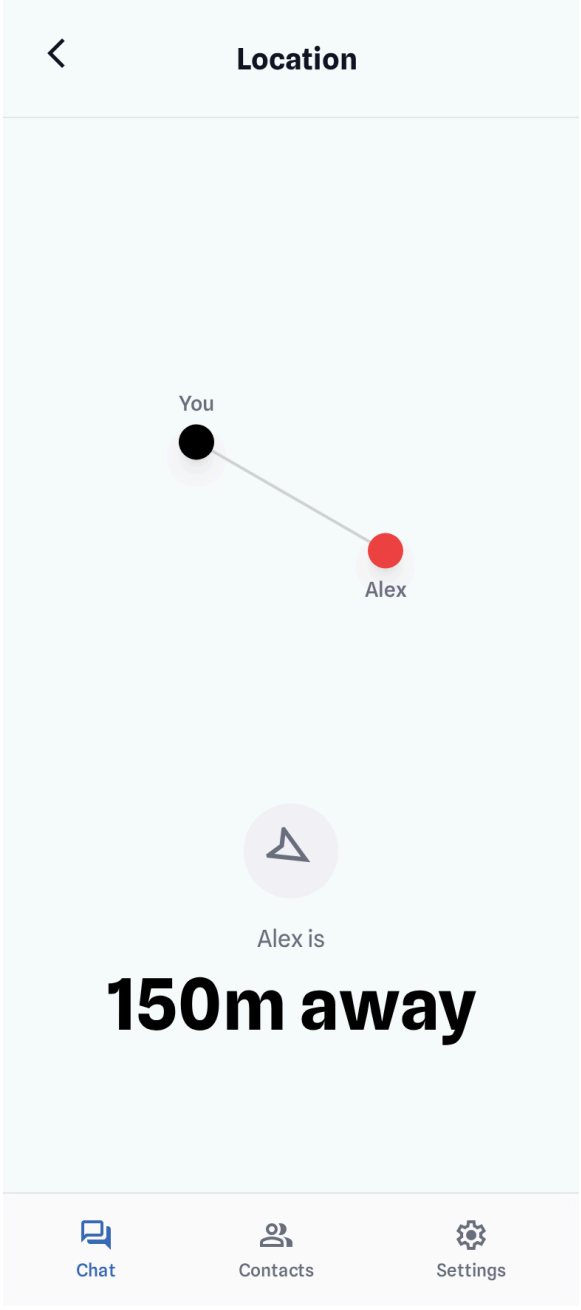
- Add Room dependencies to your `build.gradle`.
- Define your `Message` and `Conversation` data classes as `@Entity` with appropriate `@PrimaryKey` and `@ColumnInfo` annotations.
- Create `@Dao` interfaces with methods for inserting, querying, and deleting data (e.g., `@Insert`, `@Query`, `@Delete`).
- Create your `AppDatabase` class extending `RoomDatabase`.
- Access the database using coroutines for asynchronous operations.

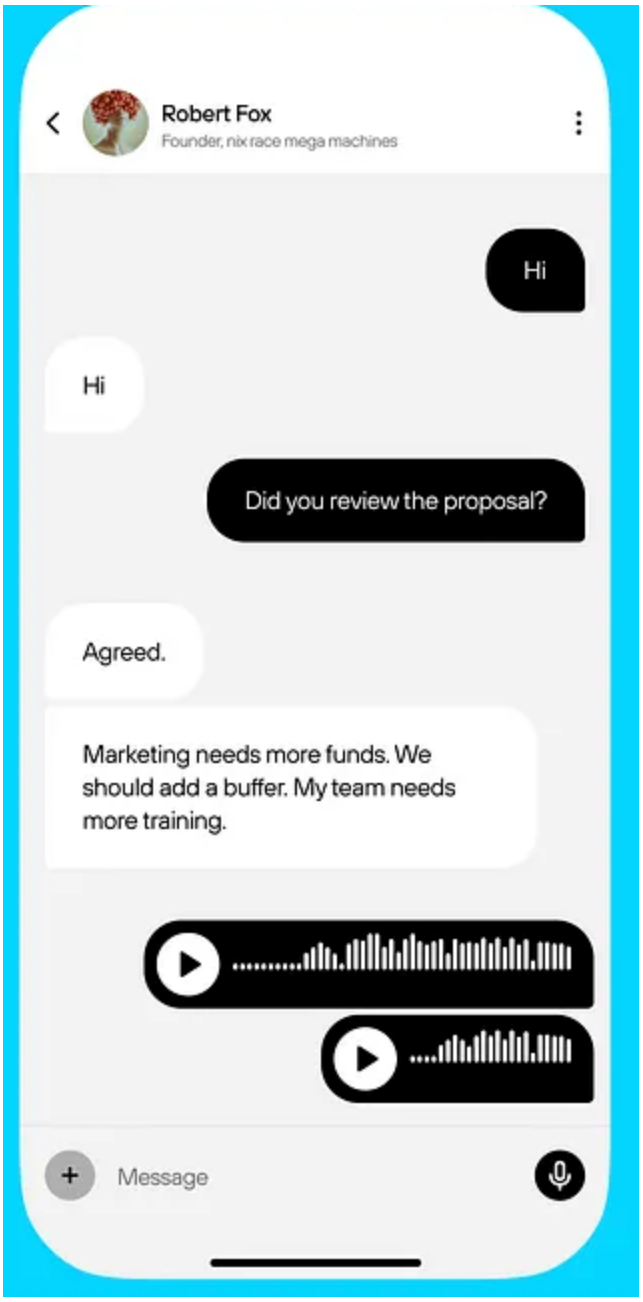
6. User Interface (Jetpack Compose)

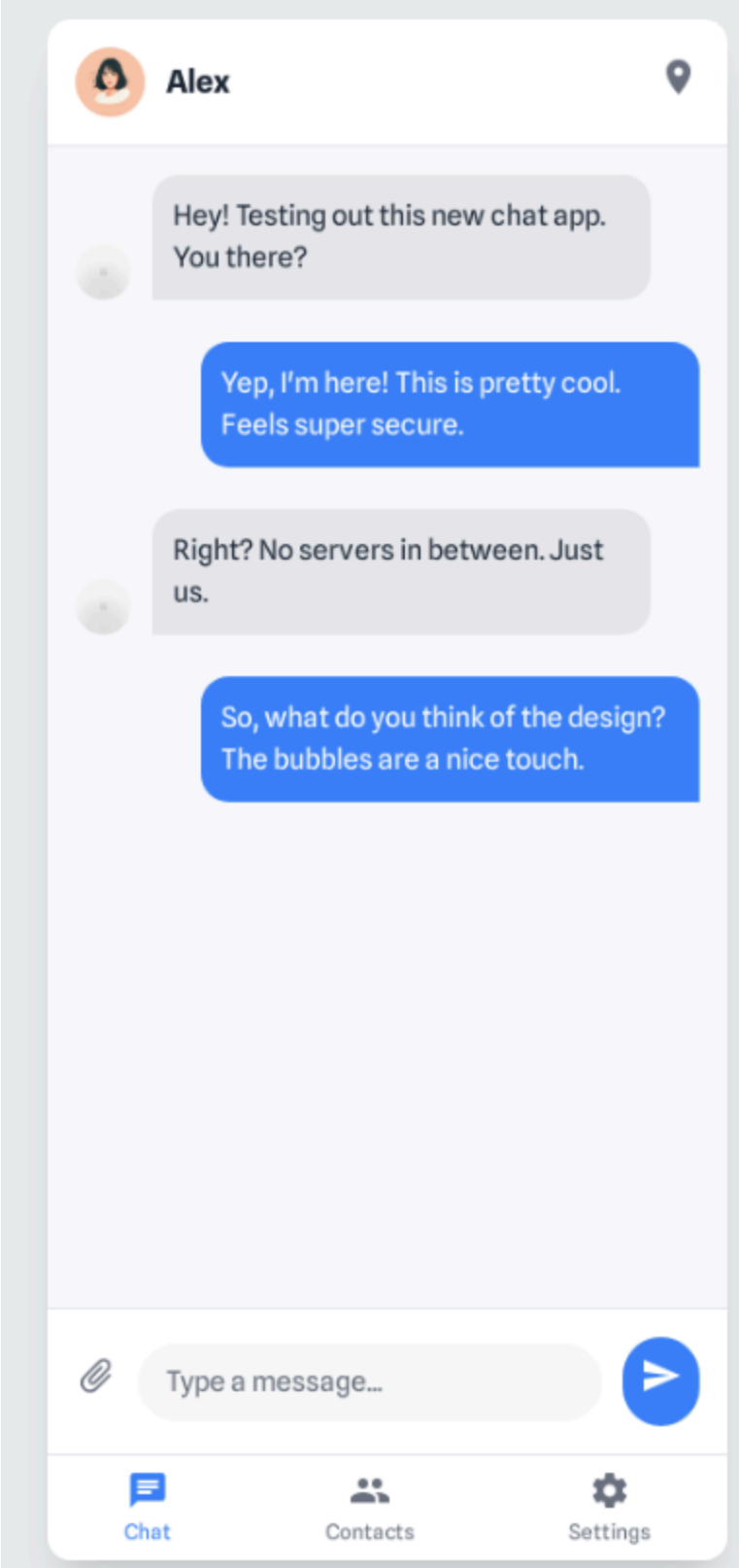
- **Scaffold:** Use `Scaffold` for basic screen layout (top app bar, bottom navigation bar, content).

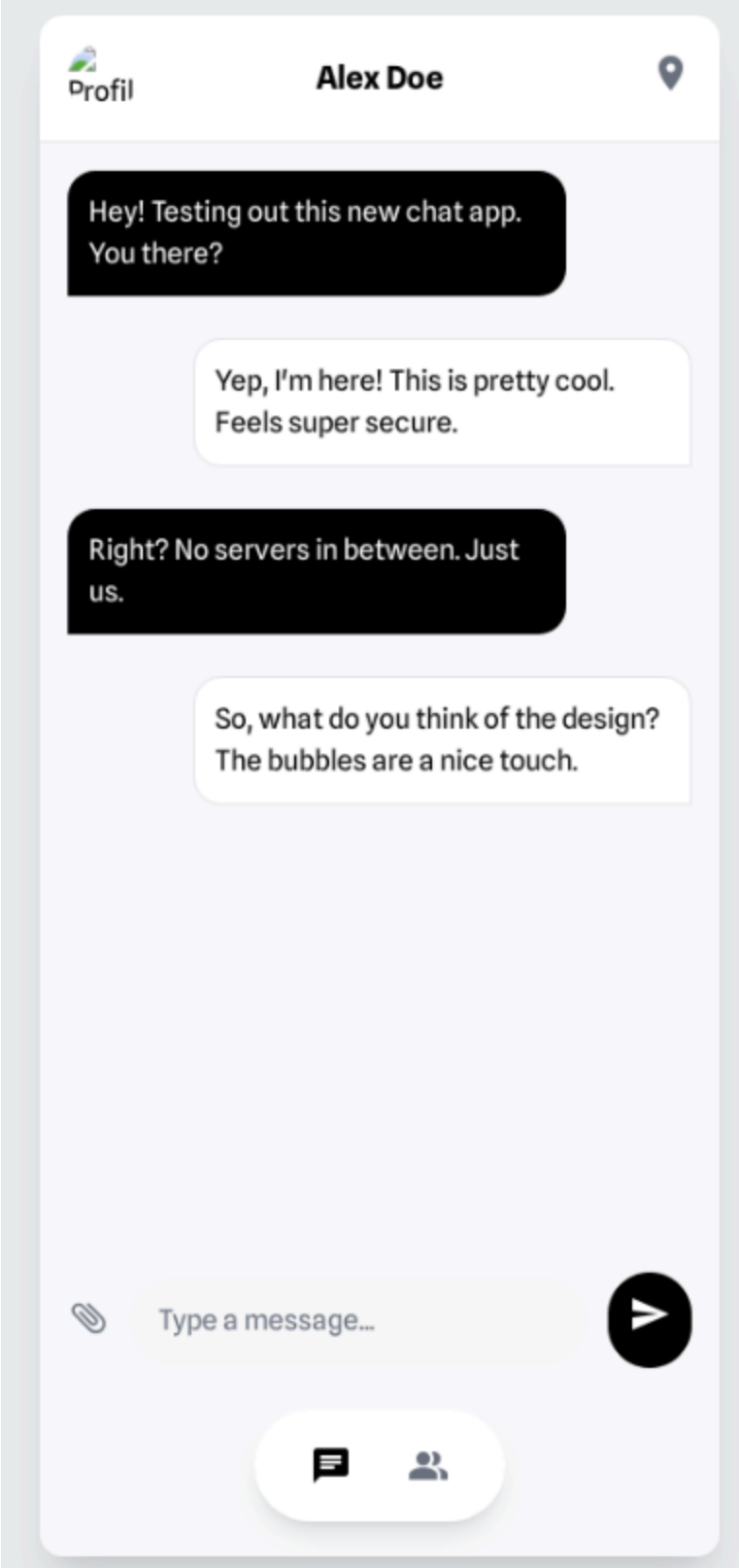
- **Bottom Navigation:** Use `NavigationBar` and `NavigationBarItem` for your "Connect," "Chats," and "Security" tabs.
- **LazyColumn:** For efficient display of long lists like devices and chat messages.
- **State Management:** Use `remember` and `mutableStateOf` for UI-specific state, and collect `StateFlow` or `LiveData` from your `ViewModels` using `collectAsStateWithLifecycle()` to drive your Composables.
- **Input:** `OutlinedTextField` or `BasicTextField` for text input.
- **Dialogs/Bottom Sheets:** For permissions, security alerts, and attachment options.
- **Custom Drawing:** For the compass arrow, you'd use `Modifier.drawWithContent` or `Canvas` within your `LocationMessageBubble` Composable to draw the arrow dynamically based on sensor data and target bearing.

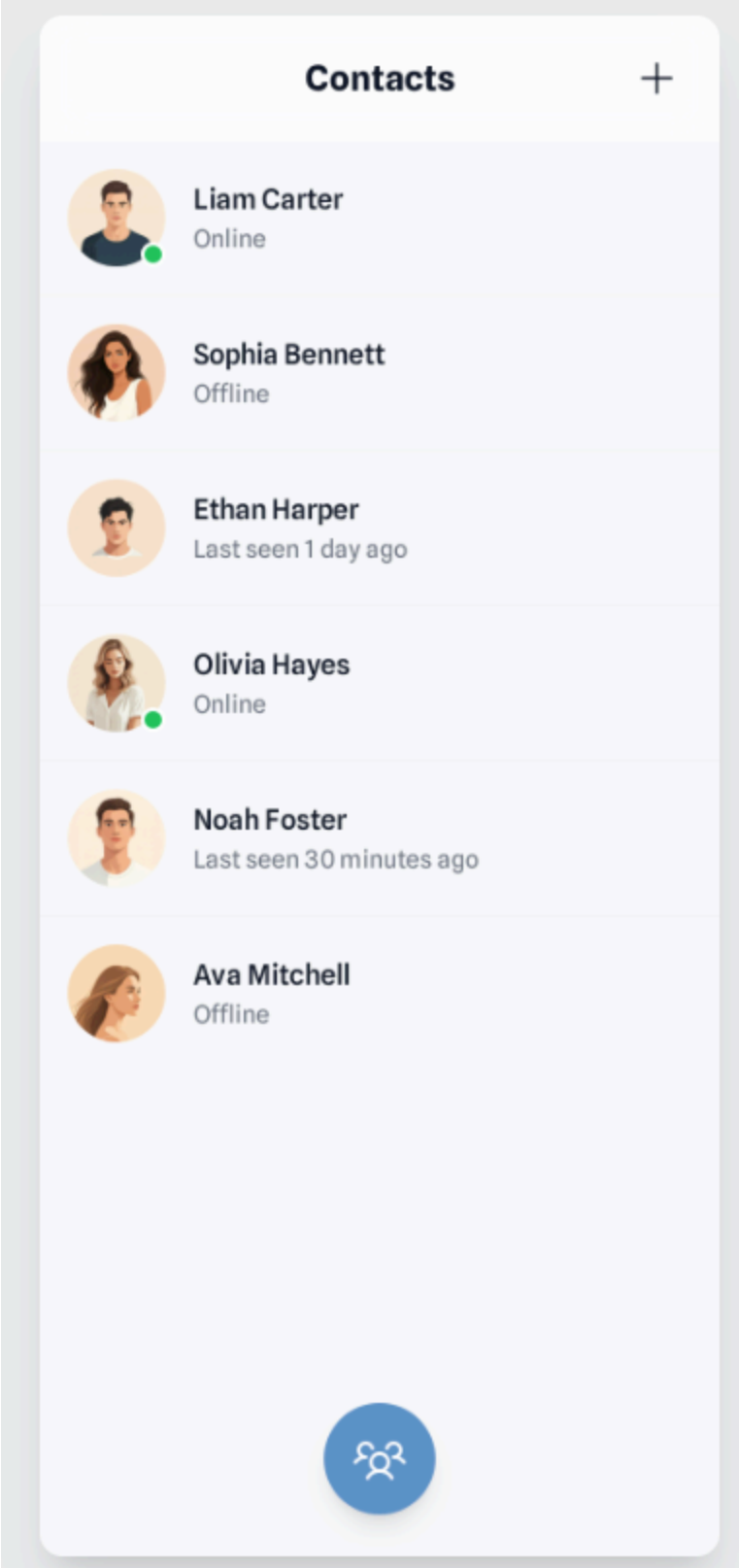
This is a high-level conceptual guide, but it outlines the exact Kotlin/Android APIs and patterns you'll use for each feature.

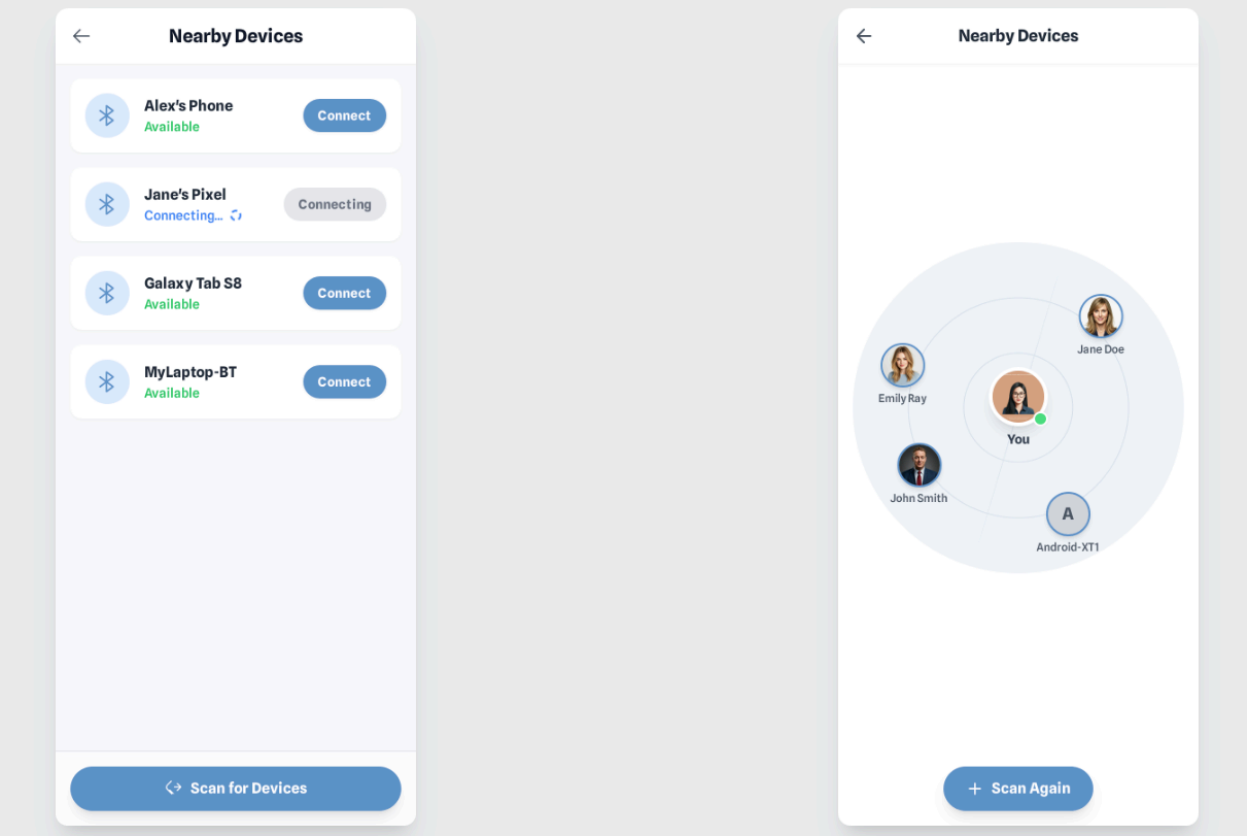

Location
You
Alex
Alex is
150m away
Chat
Contacts
Settings

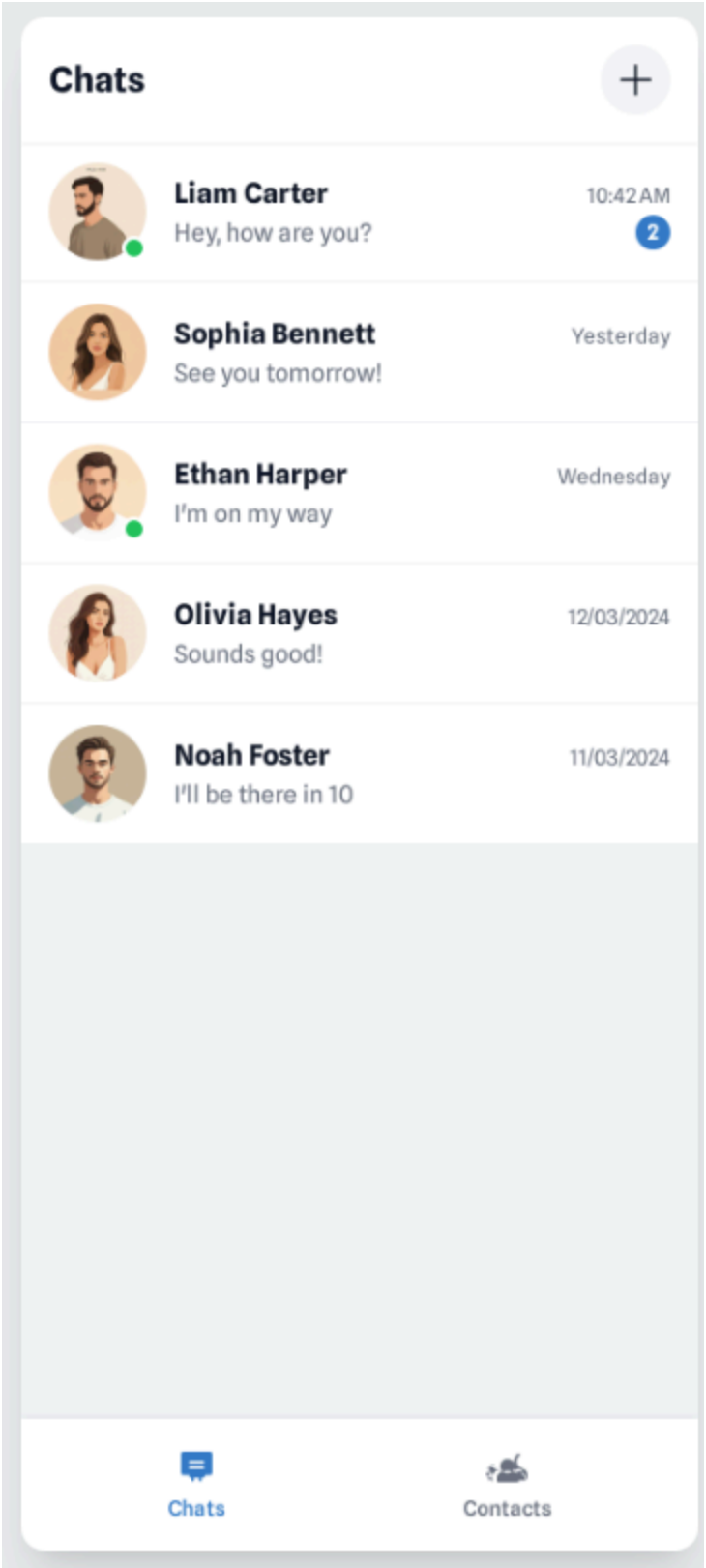

Robert Fox
Founder, nix race mega machines
Hi
Hi
Did you review the proposal?
Agreed.
Marketing needs more funds. We should add a buffer. My team needs more training.
Message

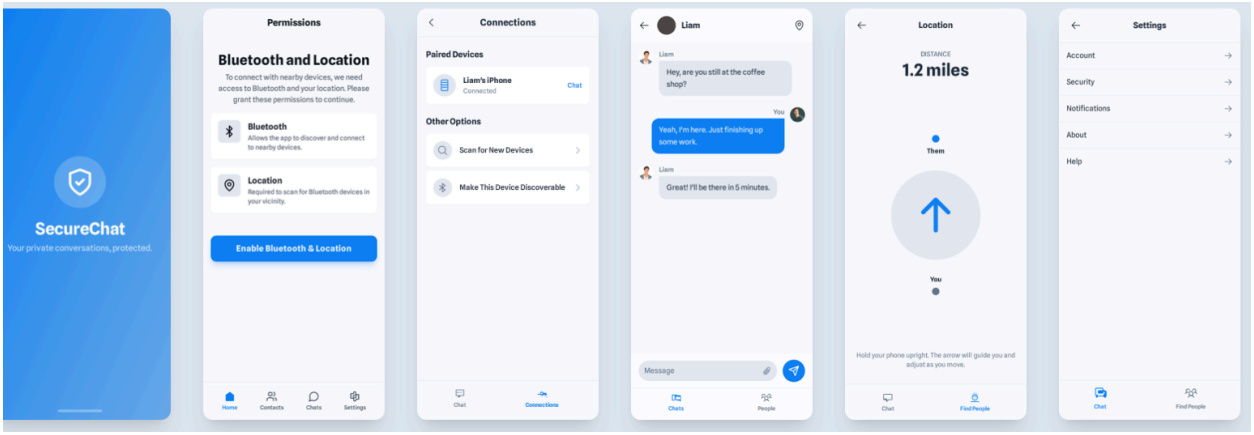

Alex
Hey! Testing out this new chat app. You there?
Yep, I'm here! This is pretty cool. Feels super secure.
Right? No servers in between. Just us.
So, what do you think of the design? The bubbles are a nice touch.
Type a message...
Chat
Contacts
Settings


Profil
Alex Doe
Hey! Testing out this new chat app. You there?
Yep, I'm here! This is pretty cool. Feels super secure.
Right? No servers in between. Just us.
So, what do you think of the design? The bubbles are a nice touch.
Type a message...


Contacts
Liam Carter
Online
Sophia Bennett
Offline
Ethan Harper
Last seen 1 day ago
Olivia Hayes
Online
Noah Foster
Last seen 30 minutes ago
Ava Mitchell
Offline


Nearby Devices
Alex's Phone
Available
Connect
Jane's Pixel
Connecting...
Connecting
Galaxy Tab S8
Available
Connect
MyLaptop-BT
Available
Connect
Scan for Devices
Nearby Devices
Jane Doe
Emily Ray
You
John Smith
A
Android-XT1
Scan Again


Chats
Liam Carter
Hey, how are you?
10:42 AM
2
Sophia Bennett
See you tomorrow!
Yesterday
Ethan Harper
I'm on my way
Wednesday
Olivia Hayes
Sounds good!
12/03/2024
Noah Foster
I'll be there in 10
11/03/2024
Chats
Contacts


SecureChat
Your private conversations, protected.
Permissions
Bluetooth and Location
To connect with nearby devices, we need access to Bluetooth and your location. Please grant these permissions to continue.
Bluetooth
Allows the app to discover and connect to nearby devices.
Location
Required to scan for Bluetooth devices in your vicinity.
Enable Bluetooth & Location
Home
Contacts
Chats
Settings
Connections
Paired Devices
Liam's iPhone
Connected
Chat
Other Options
Scan for New Devices
Make This Device Discoverable
Chat
Connections
Liam
Liam
Hey, are you still at the coffee shop?
You
Yeah, I'm here. Just finishing up some work.
Liam
Great! I'll be there in 5 minutes.
Message
Chats
People
Location
DISTANCE
1.2 miles
Them
You
Hold your phone upright. The arrow will guide you and adjust as you move.
Chat
Find People
Settings
Account
Security
Notifications
About
Help
Chat
Find People