ChatGPT

Building a Scalable Decentralized Messenger with GUN.js

Overview of GUN.js for Decentralized Apps

GUN.js is an **open-source, real-time, distributed graph database** designed for offline-first, peer-to-peer applications [1] [2]. It stores data as a graph of nodes (JSON objects) and uses CRDTs (conflict-free replicated data types) to synchronize state across peers. GUN replicates data cryptographically and in real time, so each user's app keeps a local copy of relevant data and merges updates whenever peers connect [2] [3]. Because GUN is "offline-first", data is cached locally (e.g. in browser localStorage [4]) and is available even if the network is down. In practice, browsers communicate via WebRTC or WebSocket with minimal signaling servers, so data flows directly between clients whenever possible [5]. In short, GUN provides a Firebase-like API but without any single centralized server; the global state is the union of all peers' graphs [2] [5].

To secure this peer-to-peer graph, GUN includes the **SEA (Security, Encryption, Authorization)** module, which uses the native WebCrypto API [6]. SEA lets each user create a public/private keypair (derived from their password) for authentication and encryption. For example, `db.user().create(username, password)` generates a keypair tied to that username, and `db.user().auth(username, password)` logs them in. The public key (and alias) is shared so others can verify the identity, while the private key stays secret in the browser. All messages or data can be encrypted with SEA before putting them into the GUN graph. Thus GUN natively supports end-to-end encryption: only recipients with the proper keys can decrypt the contents [6] [7].

User Profiles and Authentication

In a GUN app, **user profiles** and login can be managed entirely on the client using SEA. When a user signs up or logs in, the code might look like:

```
import GUN from 'gun';
import 'gun/sea';  // enable SEA
const db = GUN();
const user = db.user().recall({ sessionStorage: true });
user.create(username, password, err => { /* handle error or auto-login */ });
```

This creates a SEA keypair and stores it in localStorage (surviving page reloads) [4]. Once authenticated, you can store profile fields (like `user.get('alias').put(username)`) under the user's SEA-secured node. Other peers can look up a user by their public key and read their "alias" (username) or avatar. For example, one tutorial uses `user.get('alias').on(v => username.set(v))` to keep the UI showing the current user's name [8]. Because the private key never leaves the user's browser, the password is never sent over the network – it only serves as a seed to unlock the keypair. [9] 【46†】

Figure: Example login screen for a GUN.js-based chat app (user enters alias/password to create or access an account).

Using GUN's SEA means authentication is **decentralized**: there is no central auth server. Each user self-generates keys, and trust comes from knowing their public key out-of-band (e.g. via friend invite links). In fact, some projects (like Iris chat) use *no signup at all*: users simply generate a new keypair in-browser, and then exchange chat "invites" (links containing public keys) [10]. This removes any registration cost. However, it also means you must securely back up your keys (losing your private key means losing access to all your encrypted data).

Chat Data Model: One-on-One and Group Chats

Once users exist in GUN, **messages** are just data written into the shared graph. A common pattern is to dedicate a node (or set of nodes) for each conversation or "chat room". For example, a simple scheme is:

- **One-on-one chat**: Have a GUN node `chat/{id}`, where `{id}` is a unique room ID (perhaps derived from the two users' public keys). Both users read and write to `db.get('chat').get(id)`.
- **Group chat**: Similarly, define a room ID for the group, and let all members read/write that node. Alternatively, maintain a membership list so only authorized peers subscribe.

Within each chat node, each message can be stored under a timestamp or sequential key. In code, sending a message might do something like:

```
const index = new Date().toISOString();
const cipher = await SEA.encrypt(text, sharedKey);
db.get('chat').get(index).put({ author: user.is.pub, body: cipher });
```

Here the message object (with the encrypted text) is put into `chat/<timestamp>`. Other peers have subscribed to the same path and will receive the update. For example, one Svelte/GUN demo does:

```
const message = { who: await db.user(data).get('alias'), what: await
SEA.decrypt(data.what, key) };
db.get("chat").get(index).put({ what: secret });
```

and retrieves new messages via `db.get('chat').map().once(...)` [11].

Because GUN is graph-based, you can model chats flexibly. You might even store chats in user nodes (e.g. `user.get('chats').get(chatId)`) or attach metadata (timestamps, edited flags) as subfields. GUN's indexing makes it easy to query recent messages or search by index. In any case, **messages can flow immediately between peers** without a central server: when one user publishes `chat/123->timestamp`, others subscribed to `chat/123` instantly see the new data via GUN's real-time sync. (Behind the scenes, GUN CRDT merging ensures everyone eventually agrees on the history even if users go offline and reconnect later [2] [3].)

Figure: Example chat conversation in a GUN-powered app (messages appear instantly as peers sync).

Real-Time Presence and Typing Indicators

Because GUN updates propagate instantly, you can build rich real-time features on top. For **online presence**, a common approach is to have each client write a heartbeat timestamp to their own user node (e.g. `user.get('status').put({ lastSeen: now, online: true })`). Other peers read this field on that user's profile. If updates stop (e.g. the app is closed), peers will notice the absence of a recent timestamp and mark the user offline. The GUN developer community notes that you can implement presence by simply pinging the user's node with a current timestamp, which others subscribe to [12]. GUN also emits low-level events `gun.on('hi', peer)` and `gun.on('bye', peer)` when WebRTC/WebSocket peers connect or drop, which you could use for tracking network topology, though for user status it's simpler to handle it at the application level.

A **typing indicator** can be done similarly: when a user starts typing, write `chat/chatId.typing[userId] = true` (with a short timeout or clear on send); when they stop, set it to false. All clients subscribed to `chat/chatId.typing` will see who is typing. Alternatively, each client could emit a presence-status sub-object to the chat node. These ephemeral flags will automatically propagate through GUN's realtime mesh.

Media Sharing (Text, Files, Images, Video)

Text messages (strings) fit naturally into GUN's JSON model, but large binary media (images, videos, attachments) require special handling. Browsers impose a localStorage limit (~5–10 MB), so GUN is **not ideal for storing big blobs** directly [13]. The recommended best practice is to use a separate file storage or peer-to-peer file system, and store only references in GUN. For example:

- **IPFS**: Upload the file (or its encrypted chunks) to IPFS or a similar P2P file system. IPFS returns a content hash (CID). Then `gun.get('chat').get(msgId).put({ fileCid: '<cid>' })`. Clients retrieve the CID from GUN and then fetch from IPFS. (Note: you may need to manage an IPFS node or pinning service so content persists.)
- **S3 / Cloud storage**: In hybrid mode, your app could upload images to Amazon S3 or a CDN. When the upload completes, you write the URL to GUN. This offloads heavy data to a scalable host while GUN just carries metadata (usernames, file links).

The GUN community confirms this hybrid approach. One user on StackOverflow summarized: *"Gun isn't meant for file storage... I would recommend using IPFS for file storage and GUN to store the links to those files."* [13]. In practice, you might even split very large files into chunks and store each chunk (either in GUN or IPFS). Mark Nadal (GUN's creator) has an example uploading 20MB files by splitting them manually; future versions of GUN aim to automate chunking. Regardless, a hybrid solution (GUN + IPFS/S3) keeps your messaging infrastructure cost near zero, as the heavy lifting is offloaded to decentralized or pay-per-use storage.

End-to-End Encryption (E2EE)

Strong encryption is crucial. GUN's SEA lets you encrypt any data before putting it in the graph, achieving E2E security. A typical flow: when sending a message, first do `const cipher = await SEA.encrypt(messageText, sharedKey)`. Store `cipher` in GUN (as in the example above). The recipient retrieves that cipher from GUN and does `SEA.decrypt(cipher, sharedKey)` to recover the text. Because only the intended users know the secret key, the message remains confidential. In one demo, every message was encrypted client-side and only the **ciphertext** was stored under `chat/…` nodes [11].

SEA uses the WebCrypto API (ECDSA, AES, etc.) under the hood [6]. Each user has their own keypair; for one-on-one chat, you could encrypt the message with the recipient's public key. For group chat, you might derive or distribute a shared symmetric key among all members. It's up to your design: you could encrypt each message individually per recipient, or use a group key. (Some libraries like Signal or Olm/Megolm implement sophisticated group key management, but with GUN you'd need to code that or use a community solution.)

In any case, when done properly the **message content and attachments are E2E-encrypted**. However, be aware that metadata is generally **not** encrypted on a decentralized network: everyone can see who wrote to which chat node and when. For example, one developer noted *"Messages are e2e encrypted, but message timestamps currently aren't… In a decentralized network the timestamp and message author are visible to anyone."* [7]. In summary, use SEA for confidentiality, and accept that things like `db.get('chat').get(index)` edits (the fact of a message and its time) are observable by peers.

Decentralized Architecture and Fallbacks

A true decentralized messenger means **no single point of failure**. In an ideal GUN network, every client talks to other peers directly in a mesh. However, in practice you usually run at least one or a few **relay servers** to ensure connectivity. These are just standard Node.js/GUN instances (or any server running GUN) with fixed HTTPS endpoints. Clients connect to them via WebSockets or HTTPS polling, so even if peer-to-peer connections fail (due to NAT, no STUN, etc.), the server-relays will ferry updates.

GUN's documentation and community emphasize that *default/relay servers act as backup and propagation checkpoints* [14]. You might run a handful of GUN servers (e.g. on cheap cloud VMs or free-tier Heroku) to be the backbone. These servers need very little capacity – they just forward graph updates – so scaling costs stay low. For example, one tutorial suggests spinning up a **free Heroku dyno** as a GUN peer ("relay") for your app – which costs \$0 on the free plan [10]. Because Heroku filesystems are ephemeral, the author advises attaching an S3 adapter to persist the GUN data if needed [15], but even without persistence the in-browser caches keep chats alive.

Thanks to GUN's offline-first design, if one relay goes down, any other peer with a copy of the data can serve it. Each browser caches relevant data locally (in IndexedDB/localStorage), so the app remains usable offline or even if the servers die [16] [4]. When network returns, the client simply resynchronizes. In fact, GUN can even replicate via storage adapters: in Node.js you could configure GUN to use an S3 or IPFS backend so that if a client ever loses its data, it can rehydrate from those backups [17] [14].

Tools, Infrastructure, and Cost Considerations

To achieve **near-zero scaling costs**, the strategy is to leverage user devices and free/cheap services. For the client side, you can host your web app's static files on any CDN or static host (e.g. GitHub Pages, Netlify, Vercel) – often these have generous free plans. The frontend JavaScript includes the GUN library, SEA, and any UI framework (React, Svelte, etc.).

For servers (optional): a couple of small VPS or free dynos running Node.js with `gun` installed can serve as the relay nodes. For example, deploy a simple script like `const gun = require('gun')();` on a Heroku or Fly.io app. Since you only pay (or not) for very low compute, this is negligible. The tutorial above points out that one free Heroku dyno was sufficient to relay chat for an example app [10] [15]. You could also use cheap cloud functions or containers as needed.

For **storage**: use adapters as mentioned. In Node environments, Gun's adapters allow writing data to disk, LevelDB, Radix tree (Radisk), S3, or even IPFS/IPLD [17] [14]. For example, you could attach an Amazon S3 bucket so that all your GUN server data is mirrored to S3 (almost free for infrequent writes). Alternatively, clients could pin chat content to IPFS, with negligible running cost (IPFS can run in-browser or as a lightweight node). The key point is: by offloading large files (S3/IPFS) and using mostly client resources, operational cost is minimal.

Finally, you'll likely want WebRTC STUN/TURN servers to bootstrap peer connections. There are public STUN servers (like Google's free STUN) that work out of the box. TURN servers (for relay of media streams) aren't strictly needed for text chat, but if you add voice/video, you may use services like Twilio or self-hosted coturn – though voice/video is beyond GUN's scope.

Alternative Technologies

While GUN.js is one option, other decentralized chat frameworks/protocols exist:

- **OrbitDB / IPFS:** OrbitDB is a peer-to-peer database built on IPFS (using libp2p for networking) [18]. Like GUN, it offers CRDT-based logs and maps. You could store chat messages in an OrbitDB log and have peers sync via IPFS PubSub. However, IPFS networking currently requires an IPFS node (browser or daemon) and may have higher latency for small updates.
- **Secure Scuttlebutt (SSB):** A log-based P2P protocol (used by Patchwork, Manyverse). Each user has an append-only feed of their messages, which peers gossip. SSB guarantees eventual consistency but uses a custom P2P discovery mechanism. It is fully decentralized but less "web-friendly" since it requires running a local SSB server.
- **Matrix:** An open standard for decentralized chat. Matrix is server-based but federated: users sign up on a "homeserver" (could be self-hosted), and servers federate messages. It supports E2E (Olm/Megolm). It is a more heavyweight ecosystem (Synapse servers, dedicated clients like Element), and requires running your own server to avoid centralized providers.
- **Y.js / Automerge:** These are CRDT libraries for real-time collaboration. On their own they lack persistence or networking, but paired with modules (y-webrtc or PeerJS) they can form a peer mesh. You'd have to build messaging logic yourself. They excel at rich shared state (docs, editors) but could be adapted to chat.
- **Libp2p / IPFS PubSub:** Underlying libraries like libp2p (used by both GUN and OrbitDB) can be used directly to build P2P chats. For example, two browsers can join the same PubSub topic and exchange messages. But then you must manage history yourself (e.g. writing to localStorage or an external DB).
- **Others:** There are specialized P2P chat apps like Tox, Briar (Android), Session, etc. These are end-user apps, not libraries. For a web app, GUN or OrbitDB are among the few true browser-friendly P2P DBs.

The 3Box research compares GUN to OrbitDB and SSB: GUN uses a state-based CRDT without an append-only log [3], and runs peer connections over WebRTC/WebSocket [5]. OrbitDB uses IPFS/libp2p and log-CRDTs [18]. Secure Scuttlebutt is log-based. Each approach has trade-offs in simplicity, latency, and network assumptions.

Architectural Patterns and Best Practices

Based on community experience, here are some best practices and patterns for a GUN-based messenger:

- **Data Modeling:** Design a clear graph schema. E.g., have a top-level `user/{pubkey}` node with subfields for alias, avatar, contacts. Have `chat/{chatId}` nodes for conversations. Keep nodes small. Use chronological keys (ISO timestamps) or monotonic counters for messages so they naturally sort. Use `.map()` queries sparingly on large data (you can filter by time, limit results).

- **Encryption:** Always encrypt private content on the client. Use strong random keys or SEA's built-in key generation. For one-on-one chats, a shared secret can be derived (e.g. ECDH between keys). For group chats, consider rotating group keys for perfect forward secrecy. Revoke keys or rotate on membership change if security needs demand it.
- **Presence Heartbeats:** Don't spam the network. Instead of writing presence on every keypress, write it on a timer (e.g. every 30 seconds) or on connect/disconnect events. Peers can mark "online" if `now – lastSeen < threshold`.
- **Typing Flags:** Write short-lived flags (e.g. `{typing: true}`) under the chat node. For example: `db.get('chat').get(id).get('typing').get(myUserId).put(true)` when typing starts, and put `false` or delete it after a few seconds or on send. This will propagate instantly.
- **Rate Limiting / Throttling:** If the app is very active, consider debouncing frequent writes (typing, scroll events). Gun can handle a lot, but flooding with many writes (like continuous mouse moves) is wasteful.
- **Offline Caching:** Encourage clients to cache chat history. GUN does this automatically in localStorage. On login, you may load recent messages immediately from local cache while waiting for peers to sync.
- **Fallback Servers:** Run at least two reliable peers (e.g. digitalocean droplets or Heroku) so that clients have a stable relay. Using HTTPS with valid TLS certs is recommended for security (Gun over HTTPS/WebSocket).
- **Key Recovery:** Provide a way for users to back up/recover keys. For example, export the SEA keys to a file or allow mnemonic seed phrases. If a user loses their keys, all encrypted chats are lost.
- **Content Moderation / Abuse Prevention:** Decentralized networks make moderation hard. If it's private chat, trust is between friends. If you have public chatrooms, consider implementing blocklists or content filtering on the client side.
- **Versioning:** If you update data schemas, handle migration. GUN is schemaless, but both old and new clients may read the same data. Document how fields are interpreted.
- **Security:** Audit the client code thoroughly. Use HTTPS and CORS policies if needed. Because peers connect directly, be mindful of exposing any unwanted interfaces.

Challenges and Limitations

Building a P2P chat has unique challenges:

- **Conflict Resolution:** While CRDTs handle most merge conflicts automatically, edge cases can occur (e.g. two users editing the same profile field at once). Test scenarios where peers go offline and come back to sync.
- **Key Management:** Distributing and rotating encryption keys can be complex. There's no built-in key exchange protocol in basic GUN. You may need to roll your own or adapt existing ones (e.g. use SEA's pair and sign APIs).
- **Data Permanence:** Once data is written and propagated, it's hard to delete (especially if many peers have it). There's no "right to be forgotten" unless all peers agree.
- **Network Dependencies:** Though decentralized, your app is still partially dependent on network reliability. Users with extremely poor connectivity might find delays in syncing. Using fallback servers helps mitigate this.
- **Privacy:** As noted, metadata is exposed. Also, IP addresses of peers may be discoverable at the WebRTC layer unless using TURN. Decide what level of anonymity is required.
- **Scalability:** In theory, every new message is broadcast to all subscribers. For very large public chats (hundreds+ users), consider partitioning data or using a dedicated pubsub topic. GUN's

mesh can handle moderate groups, but performance degrades if thousands of peers sync all messages.

- **Security of Endpoints:** If you run your own peers, secure them (keep Node and GUN packages updated). Malicious users on the network can attempt to inject data, but they can only affect their own keyspace (they can't fake another user's signature without the private key).
- **User Experience:** The lack of central servers means initial setup may feel different (e.g. exchanging keys or invite links). Be prepared to guide users through "share your public ID". UX design is key.

Conclusion

By combining GUN.js with modern web APIs, it is entirely possible to build a **feature-rich web messenger** that is mostly serverless and cost-effective. GUN provides the backbone: a decentralized graph DB with built-in user auth (SEA), real-time sync, and offline caching. Standard chat features (profiles, 1:1 and group chats, media, presence, typing indicators) can all be implemented on top of this graph. Encryption can be handled by SEA to achieve end-to-end confidentiality.

To keep costs near zero, one should rely on peer-to-peer connectivity as much as possible, and use minimal fallback infrastructure (a couple of free-tier relay servers, and decentralized storage like IPFS or cheap S3 for media). The architecture is hybrid: *decentralized-first*, with optional servers acting only as backups or rendezvous points [14] [10]. This reduces operational load while preserving reliability.

In exploring alternatives like OrbitDB (IPFS), Matrix, or Secure Scuttlebutt, one finds similar trade-offs (e.g. federated servers vs pure P2P) [5] [18]. GUN's approach, however, is uniquely browser-friendly and graph-structured. With careful design of data models and attention to encryption and syncing details, a robust decentralized messenger is within reach.

Sources: Official GUN documentation and community tutorials [1] [4] [19] [13] [20] [5] [3]. These include deep-dives into GUN's architecture and example chat implementations, which guided the design choices above.

---

[1] [2] [6] [14] [16] Gun: An Ecosystem of Tools for Distributed Applications | by estebanrfp | Nov, 2025 | GenosDB
https://genosdb.com/gun-an-ecosystem-of-tools-for-distributed-applications-63e00b4a8fc6?gi=da51e8f77182

[3] [5] [18] 3Box Research: Comparing Distributed Databases GUN, OrbitDB, and Scuttlebutt | by 3Box Labs | 3Box Labs | Medium
https://medium.com/3box/3box-research-comparing-distributed-databases-gun-orbitdb-and-scuttlebutt-2e3b5da34ef3

[4] [17] javascript - How far does the Offline-first feature of gunJS go ? - Stack Overflow
https://stackoverflow.com/questions/50072996/how-far-does-the-offline-first-feature-of-gunjs-go

[7] [10] [15] [20] Javascript live chat tutorial: Free alternative to Firebase | Sirius Business
https://mmalmi.github.io/javascript-live-chat-tutorial-free-alternative-to-firebase

[8] [11] [19] GitHub - Envoy-VC/gun-chat: Decentralized Chat App using GUN.js and Svelte
https://github.com/Envoy-VC/gun-chat

[9] Build a Decentralized Chat App with Gun.js
https://www.toolify.ai/ai-news/build-a-decentralized-chat-app-with-gunjs-1088092

[12] Handling connect/disconnect status & get online users in GUN - Stack Overflow
https://stackoverflow.com/questions/50479815/handling-connect-disconnect-status-get-online-users-in-gun

[13] gun - How to upload and download media files using GUNDB? - Stack Overflow
https://stackoverflow.com/questions/62926913/how-to-upload-and-download-media-files-using-gundb