The Shared Table

How vault storage works, from personal namespaces to shared sandboxes

February 2026

A vault needs a database

Strip away the cryptography, the messaging layer, the enclaves, and what is left? A program that receives events, acts on them, and remembers things. That program needs a database.

Not a specific database. Not a particular vendor or schema. Just persistent, encrypted storage. The vault provider picks the implementation. It could be SQLite. It could be something else entirely. What matters is not the technology. What matters is the pattern: the vault manager receives events from the messaging layer, reads and writes to the database, and puts events back on the messaging layer. That is the entire runtime model. Everything else is organization.

But the vault manager is not a dumb pipe. When an event arrives, the vault manager reads it, determines which event handler should process it, and invokes that handler. A payment.request event triggers the payment handler. A document.sign event triggers the signing handler. An identity.verify event triggers the verification handler. The handlers are programs built and deployed by the vault provider, running inside the enclave alongside the vault manager. They are what make the vault programmable — not just a data store, but an agent that can act on your behalf within the rules you have set.

This is also why vault providers can differentiate. The handlers are the capabilities. One provider builds handlers for financial operations. Another builds handlers for healthcare consent. A third builds handlers for enterprise policy enforcement. The vault manager, the database, and the messaging layer are the foundation. The handlers are where the value lives.

> *A vault is an event-driven process with a data store and a set of handlers. Events come in. The right handler is invoked. State is updated. Events go out. The handlers are the capabilities. The database is the memory. That is the foundation everything else builds on.*

Namespaces: how the database gets its shape

A single encrypted bucket would work, technically. But it would be a mess — personal data mixed with system state mixed with connection data, all in one undifferentiated pile. Namespaces are what give the vault's database structure and isolation.

Think of namespaces as rooms in a building. Each room has a door and a lock. What is inside one room is invisible to everything in every other room. The vault manager holds all the keys and knows which room to enter for which operation. But the rooms themselves are isolated.

The personal namespace

This is you. Your identity, your keys, your credentials, your personal information. The personal namespace is yours and only yours. No connection, no service, no gateway, no other vault can see into it. When you look at your vault and see "who I am" and "what I have," you are looking at your personal namespace.

Your Ed25519 identity keypair lives here. Your cryptographic keys — the Bitcoin key, the signing key, the Ethereum key — live here. Your personal profile data lives here. The metadata catalogs that tell connections what credentials you have (without revealing the credentials themselves) are generated from what is in this namespace. This is the innermost room. Nothing gets in without you.

The system namespace

This is the vault’s own housekeeping. Key material for encryption and rotation. Session state. Configuration. Audit logs. The vault manager reads and writes to this namespace as part of its normal operations. You do not interact with it directly, but it is what keeps the vault running — the plumbing behind the walls.

At this point, with just these two namespaces, you have a working vault with no connections. A secure personal data store. Useful on its own — a digital safe that only you can open, running inside a hardware-isolated enclave. But the vault becomes interesting when you connect to things.

> *The personal namespace is yours. The system namespace is the vault’s. Two rooms, two purposes, completely isolated. A vault with only these two namespaces is already a secure personal data store. Connections are what make it a network.*

A connection adds a namespace

When you connect to a service, a person, or any other entity on the vine, the connection gets its own namespace in your vault. A new room appears in the building. The connection can interact with its room — within the terms of the contract — and you can see what is in it. But the connection cannot see your personal namespace, your system namespace, or any other connection’s namespace.

This is the keychain model from the user’s perspective. The ring a service adds to your keychain — the keys, pointers, credentials, and metadata that belong to that relationship — lives in that connection’s namespace. A retailer’s namespace holds your loyalty credential and a pointer to your order history. A healthcare provider's namespace holds encryption keys for your records and a consent manifest. A friend's namespace holds the cached profile you exchanged when you connected.

Two kinds of contents

Not everything in a connection namespace is the same. Services can add two kinds of data.

Transparent data: Data and secrets that you own and control. Your encryption keys, your credentials, your consent records. You can see these, manage them, and take them with you. If you sever the connection, these stay in your vault because they are yours.

Opaque data: Data and secrets that the service uses for its own operations. Session tokens, internal references, service-specific keys. The service needs these, your vault holds them, but you do not need to understand them. If you sever the connection, you can keep or discard them.

The contract spells out which is which. And because the data lives in your vault rather than in the service's database, revocation is real: sever the connection and the service can no longer reach its namespace. The keys, the pointers, the references — they are in your vault, behind your lock.

Ten connections, ten namespaces

As your vine grows, your vault's database grows with it. Each connection adds a namespace. Ten connections means ten isolated rooms, each with its own contents, each governed by its own contract. Your personal namespace stays untouched. No connection's namespace can see into another's. The vault manager routes events to the right namespace based on which connection is making the request.

This is also why the vault stays manageable. The namespaces do not pile up into an unstructured heap. Each one is labeled, scoped to a connection, and governed by explicit terms. You can browse any namespace, see what is in it, and remove anything you want. The structure is the namespace. The governance is the contract.

> *A connection is not just a communication channel. It is a namespace in your vault — a scoped, isolated storage space that the connection can interact with and you can govern. The vault grows one namespace at a time, each one as structured and controlled as the first.*

The shared table

Everything so far is one-sided. Your vault, your namespaces, your data. The service adds a ring to your keychain, but the ring lives in your vault. The service can interact with its namespace through the connection, but the namespace is yours.

Some relationships need more than that. Some relationships need collaboration — both sides reading, writing, and working from the same data. Not two copies. Not a sync protocol. One shared space, hosted in one vault, accessible to both parties through the connection.

This is the shared sandbox.

One table, not two copies

Think of a shared office. Each person has their own desk with their own drawers — that is the per-connection namespace. But there is also a table in the middle of the room. Both people can put things on the table, read what is on the table, and work from what is on the table. There is one table. Not two copies of the table. One.

The shared sandbox works the same way. It is a namespace that lives in one vault — the vault of whoever created it — and is accessible to the other party through the connection. Both sides can read and write to it, within the terms of the contract. But there is only one copy of the data, in one vault.

This eliminates an entire category of problems. No replication. No sync conflicts. No eventual consistency headaches. No question of which copy is authoritative. The sandbox is in one vault. That vault is the source of truth. The other party accesses it through the connection in real time.

Your toys, my toys, our table

Shared does not mean everything is mutual. Each party has their own area within the sandbox — their own toys on their own side of the table. And each party can explicitly designate specific items as visible or writable to the other side. The sharing is deliberate, per-item, not all-or-nothing.

Area	Who can read	Who can write	Example
Your side	You only	You only	Your draft notes, your internal references
Their side	They only	They only	Their internal status flags, their processing notes
Mutual	Both parties	Both parties (per contract)	Signed documents, agreed terms, shared project status

You can move something from your side to the mutual area whenever you want. The other party can do the same. But nothing appears in the mutual area without someone deliberately putting it there. And the contract governs what each side can do in the mutual area — perhaps one side can read but not write, or writes require the other's approval.

Who holds it

The vault that creates the shared sandbox hosts it. This is a simple rule that avoids ambiguity: whoever initiates the collaboration holds the table. The other party accesses it through the connection.

This matters most when the collaboration ends. If the connection is severed, the creator keeps the sandbox. The other party loses access. If the contract allows it, the other party can take a copy before disconnecting. But there is no question of ownership — the creator holds the original, and the contract spells out what happens to the data when the relationship ends.

Example: Your employer creates a shared sandbox for a project. You both work in it throughout the engagement. When you leave the company, the employer keeps the sandbox — it is their project. If the contract includes a portability clause, you can export a copy of the mutual items before the connection is severed. Your side of the sandbox — your draft notes, your personal references — was always in your per-connection namespace, not in the shared sandbox. That stays with you regardless.

Any party can initiate a shared sandbox. It is just another aspect of the connection contract. The terms dictate who hosts it, what each side can do in it, and what happens when the connection ends. The same consent model that governs everything else on the vine governs the shared sandbox.

> *The shared sandbox is one table, in one vault, accessible to both parties. The creator hosts it. The contract governs it. Both sides have their own area plus a mutual area. When the collaboration ends, the creator keeps the original. No replication. No sync conflicts. No ambiguity.*

What this looks like

The progression from personal namespace to shared sandbox is the progression from a safe to a workspace. Each step adds capability without losing control.

Doctor and patient

A patient connects to a healthcare provider. The connection creates a namespace in the patient's vault (the keychain ring: encryption keys for medical records, a consent manifest, a pointer to the patient's records in the provider's system) and a namespace in the provider's vault (the connection credentials, contract terms, request logs).

The provider creates a shared sandbox for the treatment relationship. The provider's side of the sandbox holds clinical notes, treatment protocols, and internal scheduling references. The patient's side holds symptom logs, medication notes, and personal health observations. The mutual area holds the agreed treatment plan, signed consent forms, and shared test results that both sides need to reference.

The patient can see everything in the mutual area and everything on their side. They cannot see the provider's clinical notes unless the provider moves them to the mutual area. The provider can see everything in the mutual area and everything on their side. They cannot see the patient's personal notes unless the patient shares them. Both sides work from the same treatment plan without either side having full visibility into the other's internal thinking.

Employer and employee

An employer creates a shared sandbox for a project team. The employer's side holds project assignments, budget references, and internal HR flags.

The employee’s side holds time logs, personal task notes, and draft work. The mutual area holds approved deliverables, signed agreements, and shared project status.

The sandbox lives in the employer’s vault because the employer created it. When the employee leaves, the employer keeps the project sandbox. The employee keeps their per-connection namespace (their keychain ring with credentials and personal references) and can export mutual items if the contract allows. The employee’s personal notes on their side of the sandbox are gone — the employer holds the sandbox, but the contract can require that the employee’s side be purged on disconnection, or that the employee receives a copy before severance.

Two parties in a transaction

A buyer and seller enter a transaction. Either party can create the shared sandbox — the contract terms determine who hosts it. The buyer’s side holds payment credentials and internal approval status. The seller’s side holds inventory references and fulfillment status. The mutual area holds the agreed terms, the transaction state, and signed confirmations.

As the transaction progresses, both sides update the mutual area: the buyer confirms payment, the seller confirms shipment, both sign off on completion. The entire transaction history lives in one place, in one vault, with no sync conflicts and no question about which version is current. When the transaction completes, the host keeps the record. If a dispute arises, both sides can reference the mutual area — it is the same data, not two divergent copies.

The full picture

The vault's database tells the story of your digital life in namespaces.

Namespace	What it holds	Who can access
Personal	Your identity, keys, credentials, personal data	You only. No connection can see in.
System	Vault configuration, key material, session state, audit logs	Vault manager only. Housekeeping.
Connection (per)	Keys, pointers, credentials, metadata for that relationship	You and that connection, per contract terms.
Shared sandbox	Each side's private area plus a mutual area for collaboration	Both parties, per contract. Creator hosts.

Each layer builds on the one before it. The personal namespace is the foundation. The system namespace is the plumbing. Connection namespaces are the keychain rings. The shared sandbox is the collaborative workspace. And all of it — every namespace, every item, every access rule — is governed by the same principles: isolation by default, access by contract, visibility under your control.

Vault providers can innovate at every layer. The database engine, the indexing strategy, the performance optimizations, the user interface for browsing namespaces — all of this is implementation space where providers can differentiate. What remains constant is the namespace model and the contract-based governance. The rooms have doors. The doors have locks. You hold the master key.

> *A vault starts as a personal safe. Add a connection and it becomes a keychain. Add a shared sandbox and it becomes a workspace. The progression is natural, the isolation is structural, and the control is always yours. The database is just rooms, doors, and locks. The namespaces are the rooms. The contracts are the locks. And you hold the master key.*