Calimero SDK Guide for Builders

The Calimero SDK (`core/crates/sdk` and `core/crates/storage`) provides everything you need to build distributed, peer-to-peer applications with automatic conflict-free synchronization.

Overview

The SDK consists of two main components:

- **Application SDK** (`core/crates/sdk`): Macros, event system, private storage, and runtime integration
- **Storage SDK** (`core/crates/storage`): CRDT collections with automatic merge semantics

Together, they enable you to build applications with:

- Automatic conflict resolution via CRDTs
- Real-time event propagation
- Private node-local storage
- Type-safe state management

Core Concepts

State Definition

Applications define **state** using the `#[app::state]` macro:

```
use calimero_sdk::app;
use calimero_sdk::borsh::{BorshDeserialize, BorshSerialize};
use calimero_storage::collections::{LwwRegister, UnorderedMap};

#[app::state]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct MyApp {
    items: UnorderedMap<String, LwwRegister<String>>,
}
```

Key points:

- State is persisted and synchronized across nodes

- Must derive `BorshSerialize` and `BorshDeserialize` for persistence
- Use `#[app::state(emits = Event)]` to enable event emission

Logic Implementation

Implement **logic** using the `#[app::logic]` macro:

```rust
#[app::logic]
impl MyApp {
    #[app::init]
    pub fn init() -> MyApp {
        MyApp {
            items: UnorderedMap::new(),
        }
    }

    pub fn add_item(&mut self, key: String, value: String) -> app::Result<()>
{
        self.items.insert(key, value.into())?;

        Ok(())
    }

    pub fn get_item(&self, key: &str) -> app::Result<Option<String>> {
        Ok(self.items.get(key)?.map(|v| v.get().clone()))
    }
}
```

Key points:

- `#[app::init]` marks the initialization function
- Mutation methods (`&mut self`) generate deltas and sync
- View methods (`#[app::view]`) are read-only and faster
- Use `app::Result<T>` for error handling

CRDT Collections

Available Collections

Collection	Use Case	Merge Strategy	Nesting
Counter	Counters, metrics	**Sum**	Leaf
LwwRegister<T>	Single values	**Latest timestamp**	Leaf

Collection	Use Case	Merge Strategy	Nesting
ReplicatedGrowableArray	Text, documents	**Character-level**	Leaf
UnorderedMap<K,V>	Key-value storage	**Recursive per-entry**	Can nest
Vector<T>	Ordered lists	**Element-wise**	Can nest
UnorderedSet<T>	Unique values	**Union**	Simple values
Option<T>	Optional CRDTs	**Recursive if Some**	Wrapper

UnorderedMap

Key-value storage with automatic conflict resolution:

```
use calimero_storage::collections::UnorderedMap;

let mut map: UnorderedMap<String, String> = UnorderedMap::new();

// Insert value (conflict-free)
map.insert("key".to_string(), "value".to_string())?;

// Get value
let value = map.get("key")?;  // Returns Option<V>

// Remove value
map.remove("key")?;

// Check existence
if map.contains("key")? {
    // ...
}

// Iterate entries
for (key, value) in map.entries()? {
    // ...
}
```

Vector

Ordered list with element-wise merging:

```
use calimero_storage::collections::Vector;

let mut vec: Vector<String> = Vector::new();
```

```rust
// Append element
vec.push("item".to_string())?;

// Get element by index
let item = vec.get(0)?;  // Returns Option<T>

// Insert element
vec.push("first".to_string());

// Update element at index
vec.update(0, "second".to_string());

// Remove element
vec.pop();
```

Counter

Distributed counter with automatic summation:

```rust
use calimero_storage::collections::Counter;

// bool flag true allows decrement on Counter
let mut counter: Counter<true> = Counter::new();

// Increment
counter.increment()?;

// Decrement
counter.decrement()?;

// Get value
let value = counter.value()?;  // Returns i64
```

LwwRegister

Last-Write-Wins register for single values:

```rust
use calimero_storage::collections::LwwRegister;

let mut register: LwwRegister<String> =
LwwRegister::new("initial".to_string());

// Set value (latest timestamp wins)
register.set("updated".to_string());

// Get value
let value = register.get().clone();
```

UnorderedSet

Set with union-based merging:

```rust
use calimero_storage::collections::UnorderedSet;

let mut set: UnorderedSet<String> = UnorderedSet::new();

// Insert element
set.insert("item".to_string())?;

// Check membership
if set.contains("item")? {
    // ...
}

// Remove element
set.remove("item")?;
```

Mergeable Trait

Custom structs that automatically resolve conflicts when fields are updated concurrently across nodes.

```rust
use calimero_storage_macros::Mergeable;
use calimero_storage::collections::{Counter, LwwRegister, UnorderedMap};
use calimero_sdk::borsh::{BorshDeserialize, BorshSerialize};

// Zero-boilerplate custom struct with automatic merge
#[derive(Mergeable, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct TeamStats {
    wins: Counter,
    losses: Counter,
    draws: Counter,
}

#[app::state]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct LeagueManager {
    // Use TeamStats in CRDT collections
    teams: UnorderedMap<String, TeamStats>,
}

#[app::logic]
impl LeagueManager {
    #[app::init]
    pub fn init() -> LeagueManager {
        LeagueManager {
            teams: UnorderedMap::new(),
        }
    }

    pub fn record_win(&mut self, team_name: String) -> app::Result<()> {
        let mut team = self.teams
            .entry(team_name.clone())?
            .or_insert_with(|| TeamStats {
```

```
                wins: Counter::new(),
                losses: Counter::new(),
                draws: Counter::new(),
            })?;

        team.wins.increment()?;
        Ok(())
    }
}
```

How It Works

Deriving Mergeable: When you add `#[derive(Mergeable)]` to a struct, the macro generates a `merge` method that:

1. **Field-by-field merging:** Calls `merge()` on each field (Counter, LwwRegister, nested maps, etc.)
2. **Recursive resolution:** If fields are themselves Mergeable (nested structs), merges propagate down
3. **Conflict-free convergence:** All nodes converge to the same state regardless of operation order

When merge is called:

- Only during **concurrent updates** to the same root entity (rare, ~1% of operations)
- NOT on local operations (those are O(1) writes)
- NOT when different keys are updated (DAG handles those independently)

Requirements:

- All fields must implement `Mergeable` (Counter, LwwRegister, UnorderedMap, etc.)
- Struct must have named fields (not tuple struct)
- Must derive `BorshSerialize` and `BorshDeserialize` for persistence

What the macro generates:

```
impl Mergeable for TeamStats {
    fn merge(&mut self, other: &Self) -> Result<(), MergeError> {
        self.wins.merge(&other.wins)?;      // Counter merge: sum
        self.losses.merge(&other.losses)?;  // Counter merge: sum
        self.draws.merge(&other.draws)?;    // Counter merge: sum
        Ok(())
    }
}
```

Manual implementation (when needed):

```rust
impl Mergeable for TeamStats {
    fn merge(&mut self, other: &Self) -> Result<(), MergeError> {
        // 1. Merge all CRDT fields
        self.wins.merge(&other.wins)?;
        self.losses.merge(&other.losses)?;
        self.draws.merge(&other.draws)?;

        // 2. Add custom validation
        let total = self.wins.value()? + self.losses.value()? +
self.draws.value()?;
        if total > 10000 {
            return Err(MergeError::StorageError("Too many
games".to_string()));
        }

        // 3. Add logging or metrics
        eprintln!("Merged stats: {} total games", total);

        Ok(())
    }
}
```

Use Cases

Domain modeling with custom types:

- User profiles with multiple fields (name, bio, settings)
- Product records with inventory, pricing, metadata
- Game entities with stats, equipment, achievements
- Documents with title, content, tags, version info

Complex nested structures:

- Team → Members → Individual stats
- Organization → Departments → Employees → Performance metrics
- Shopping cart → Items → Variants → Quantity

Business logic enforcement:

- Validate constraints during merge (max values, allowed ranges)
- Apply domain rules (e.g., balance can't go negative after merge)
- Log merge events for audit trails

Type safety and composability:

- Encapsulate related CRDT fields into semantic types
- Reuse custom structs across multiple UnorderedMaps or Vectors
- Build hierarchies of Mergeable types (structs containing Mergeable structs)

When to Use

Use `#[derive(Mergeable)]` when:

- Creating custom types with multiple CRDT fields
- Building domain models that sync across nodes
- Need semantic grouping of related CRDTs
- Want type-safe, reusable composite types
- Standard field-by-field merge is correct for your use case

Use manual `impl Mergeable` when:

- Custom validation rules during merge
- Logging, metrics, or debugging during merge
- Business logic constraints (e.g., invariants to maintain)
- Conditional merging based on field values
- Whole-value Last-Write-Wins with timestamp ordering

Don't use Mergeable when: - Data is node-local only (use `#[app::private]` instead) - Data is immutable after creation (use `FrozenStorage` instead) - Data is user-owned and signed (use `UserStorage` instead) - Using primitives (String, u64) without wrapping in CRDTs — **this will not compile**

Common mistake:

```
// ❌ WRONG: Primitives don't implement Mergeable
#[derive(Mergeable)]
pub struct User {
    name: String,  // Compile error!
    age: u64,      // Compile error!
}

// ✅ CORRECT: Wrap in CRDTs
#[derive(Mergeable, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct User {
    name: LwwRegister<String>,  // Latest timestamp wins
    age: LwwRegister<u64>,      // Proper conflict resolution
}

// Natural usage with auto-casting:
let user = User {
    name: "Alice".to_string().into(),  // .into() → LwwRegister
    age: 30.into(),
};
let name_str: &str = &*user.name;  // Deref for access
```

Decision tree:

Your Data	Use
Custom struct with multiple CRDT fields	`#[derive(Mergeable)]`
Need custom merge validation/logging	Manual `impl Mergeable`
Node-local secrets or caching	`#[app::private]` (not Mergeable)
Immutable content-addressed data	`FrozenStorage` (not Mergeable custom struct)
User-owned, signed data	`UserStorage` (not Mergeable custom struct)
Single primitive value that syncs	`LwwRegister<T>` (already Mergeable)

Event System

Applications can emit events for real-time updates:

```rust
#[app::state(emits = for<'a> Event<'a>)]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct MyApp {
    items: UnorderedMap<String, LwwRegister<String>>,
}

// Define event types
#[app::event]
pub enum Event<'a> {
    ItemAdded {
        key: &'a str,
        value: &'a str,
    },
    ItemRemoved {
        key: &'a str,
    },
}

#[app::logic]
impl MyApp {
    pub fn add_item(&mut self, key: String, value: String) -> app::Result<()>
{
        self.items.insert(key.clone(), value.clone())?;

        // Emit event (propagated to all peers)
        app::emit!(Event::ItemAdded {
            key: &key,
```

```
            value: &value,
        });

        Ok(())
    }
}
```

Event lifecycle:

1. Emitted during method execution
2. Included in delta broadcast
3. Handlers execute on peer nodes (not author node)
4. Handlers can update UI or trigger side effects

User Storage

User-owned, signed storage collection for per-user data. Keys are PublicKeys (32 bytes) that identify the user who owns the data. Writes are signed by the executor and verified on other nodes.

```
...
use calimero_storage::collections::{UserStorage};

#[app::state(emits = for<'a> Event<'a>)]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct KvStore {
    // Simple user-owned data (e.g., a user's profile name)
    // Stores: UnorderedMap<PublicKey, LwwRegister<String>>
    user_items_simple: UserStorage<LwwRegister<String>>,
    // Nested user-owned data (e.g., a user's private key-value store)
    // Stores: UnorderedMap<PublicKey, UnorderedMap<String,
LwwRegister<String>>>
    user_items_nested: UserStorage<NestedMap>,

}
...
 /// Sets a simple string value for the *current* user.
pub fn set_user_simple(&mut self, value: String) -> app::Result<()> {
    let executor_id = calimero_sdk::env::executor_id();
    app::log!(
        "Setting simple value for user {:?}: {:?}",
        executor_id,
        value
    );

    app::emit!(Event::UserSimpleSet {
        executor_id: executor_id.into(),
        value: &value
    });
```

```
    self.user_items_simple.insert(value.into())?;
    Ok(())
}

/// Gets the simple string value for the *current* user.
pub fn get_user_simple(&self) -> app::Result<Option<String>> {
    let executor_id = calimero_sdk::env::executor_id();
    app::log!("Getting simple value for user {:?}", executor_id);

    Ok(self.user_items_simple.get()?.map(|v| v.get().clone()))
}

/// Gets the simple string value for a *specific* user.
pub fn get_user_simple_for(&self, user_key: PublicKey) ->
app::Result<Option<String>> {
    app::log!("Getting simple value for specific user {:?}", user_key);
    Ok(self
        .user_items_simple
        .get_for_user(&user_key)?
        .map(|v| v.get().clone()))
}

// --- User Storage (Nested) Methods ---

/// Sets a key-value pair in the *current* user's nested map.
pub fn set_user_nested(&mut self, key: String, value: String) ->
app::Result<()> {
    let executor_id = calimero_sdk::env::executor_id();
    app::log!(
        "Setting nested key {:?} for user {:?}: {:?}",
        key,
        executor_id,
        value
    );

    // This is a get-modify-put operation on the user's T value
    let mut nested_map = self.user_items_nested.get()?.unwrap_or_default();
    nested_map.map.insert(key.clone(), value.clone().into())?;
    self.user_items_nested.insert(nested_map)?;

    app::emit!(Event::UserNestedSet {
        executor_id: executor_id.into(),
        key: &key,
        value: &value
    });
    Ok(())
}

/// Gets a value from the *current* user's nested map.
pub fn get_user_nested(&self, key: &str) -> app::Result<Option<String>> {
    let executor_id = calimero_sdk::env::executor_id();
    app::log!("Getting nested key {:?} for user {:?}", key, executor_id);

    let nested_map = self.user_items_nested.get()?;
    match nested_map {
        Some(map) => Ok(map.map.get(key)?.map(|v| v.get().clone())),
        None => Ok(None),
```

```
    }
}
```

How It Works

Writing: When you call `insert(value)` on UserStorage, the storage layer creates an action for the current executor (user).

Signing: The action is signed using the executor's identity private key, with a signature and nonce embedded in the metadata.

Reading: - Use `get()` to retrieve the current executor's data - Use `get_for_user(&user_key)` to retrieve a specific user's data

Verification: When other nodes receive this action, they verify:

- **Signature:** Validates against the owner's public key
- **Replay Protection:** Ensures the nonce is strictly greater than the last-seen nonce

Use Cases

- Per-user settings and preferences
- User-owned game data (scores, inventory)
- Personal documents with ownership verification
- Any data that should be verifiably owned by a specific user

Private Storage

For node-local data (secrets, caches, per-node counters):

```rust
#[derive(BorshSerialize, BorshDeserialize, Debug)]
#[borsh(crate = "calimero_sdk::borsh")]
#[app::private]
pub struct Secrets {
    secrets: UnorderedMap<String, String>,
}

impl Default for Secrets {
    fn default() -> Self {
        Self {
            secrets: UnorderedMap::new(),
        }
    }
}

pub fn use_private_storage() {
    // set secret values
    let mut secrets = Secrets::private_load_or_default()?;
        let mut secrets_mut = secrets.as_mut();
```

```
        secrets_mut
            .secrets
            .insert("secret_ID".to_string(), "secret".to_string())?;
    // get secret values
    let secrets = Secrets::private_load_or_default()?;
    let map: BTreeMap<_, _> = secrets.secrets.entries()?.collect();
}
```

Key properties:

- Never replicated across nodes
- Stored via `storage_read` / `storage_write` directly
- Never included in CRDT deltas
- Only accessible on the executing node

Frozen Storage

Immutable, content-addressable storage collection. Values are keyed by their SHA256 hash, ensuring content-addressability. Once inserted, values cannot be updated or deleted.

```
...
use calimero_storage::collections::{FrozenStorage};

#[app::state(emits = for<'a> Event<'a>)]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct KvStore {
    // Content-addressable, immutable data
    // Stores: UnorderedMap<Hash, FrozenValue<String>>
    frozen_items: FrozenStorage<String>,
}
...
/// Adds an immutable value to frozen storage.
/// Returns the hex-encoded SHA256 hash (key) of the value.
pub fn add_frozen(&mut self, value: String) -> app::Result<String> {
    app::log!("Adding frozen value: {:?}", value);

    let hash = self.frozen_items.insert(value.clone().into())?;

    app::emit!(Event::FrozenAdded {
        hash,
        value: &value
    });

    let hash_hex = hex::encode(hash);
    Ok(hash_hex)
}

/// Gets an immutable value from frozen storage by its hash.
pub fn get_frozen(&self, hash_hex: String) -> app::Result<String> {
    app::log!("Getting frozen value for hash {:?}", hash_hex);
    let mut hash = [0u8; 32];
```

```
    hex::decode_to_slice(hash_hex, &mut hash[..])
        .map_err(|_| Error::NotFound("dehex error"))?;

    Ok(self
        .frozen_items
        .get(&hash)?
        .map(|v| v.clone())
        .ok_or_else(|| Error::FrozenNotFound("Frozen value is not found"))?)
}
```

How It Works

Content-Addressing: When you call `insert(value)`, the storage:

- Serializes the value
- Computes its SHA256 hash
- Returns the hash, which serves as the immutable key
- Uses the hash as the key in the underlying map

Immutability: Once inserted, values cannot be modified or deleted. The frozen storage enforces this at the storage layer.

Verification: The storage layer enforces:

- **No Updates/Deletes:** Update and delete operations are strictly forbidden
- **Content-Addressing:** Insert operations ensure the key matches the SHA256 hash of the value

Use Cases

- Audit logs and immutable records
- Document versioning (each version gets a unique hash)
- Certificates and attestations
- Content-addressable data sharing
- Deduplication (same content = same hash)

Common Patterns

Pattern 1: Simple Key-Value Store

```
#[app::state(emits = for<'a> Event<'a>)]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct KvStore {
    items: UnorderedMap<String, LwwRegister<String>>,
}
```

```rust
#[app::logic]
impl KvStore {
    #[app::init]
    pub fn init() -> KvStore {
        KvStore {
            items: UnorderedMap::new(),
        }
    }

    pub fn set(&mut self, key: String, value: String) -> app::Result<()> {
        self.items.insert(key, value.into())?;
        Ok(())
    }

    pub fn get(&self, key: &str) -> app::Result<Option<String>> {
        Ok(self.items.get(key)?.map(|v| v.get().clone()))
    }
}
```

Pattern 2: Counter with Metrics

```rust
#[app::state()]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct Metrics {
    page_views: UnorderedMap<String, Counter>,
}

#[app::logic]
impl Metrics {
    #[app::init]
    pub fn init() -> Metrics {
        Metrics {
            page_views: UnorderedMap::new(),
        }
    }

    pub fn track_page_view(&mut self, page: String) -> app::Result<()> {
        if let Some(mut counter) = self.page_views.get_mut(&page)? {
            counter.increment()?;
        } else {
            let mut counter = Counter::new();
            counter.increment()?;
            self.page_views.insert(page, counter)?;
        }
        Ok(())
    }

    pub fn get_views(&self, page: &str) -> app::Result<u64> {
        match self.page_views.get(page)? {
            Some(counter) => Ok(counter.value()?),
            None => Ok(0),
        }
```

```
    }
}
```

Pattern 3: Nested Structures

```rust
#[app::state]
#[derive(Debug, BorshSerialize, BorshDeserialize)]
#[borsh(crate = "calimero_sdk::borsh")]
pub struct TeamMetrics {
    // Map of team → Map of member → Counter
    teams: UnorderedMap<String, UnorderedMap<String, Counter>>,
}

#[app::logic]
impl TeamMetrics {
    #[app::init]
    pub fn init() -> TeamMetrics {
        TeamMetrics {
            teams: UnorderedMap::new(),
        }
    }

    pub fn increment_metric(
        &mut self,
        team: String,
        member: String,
    ) -> app::Result<()> {
        let mut members = self.teams
            .entry(team)?
            .or_insert_with(|| UnorderedMap::new())?;

        let mut counter = members
            .entry(member)?
            .or_insert_with(|| Counter::new())?;

        counter.increment()?;
        Ok(())
    }
}
```

Building Applications

Project Setup

```
# Create new Rust project
cargo new my-calimero-app
cd my-calimero-app
```

```
# Add dependencies to Cargo.toml
[dependencies]
# Use latest release or specified version
```

```
calimero-sdk =  { git = "https://github.com/calimero-network/core", branch =
"master" }
calimero-storage =  { git = "https://github.com/calimero-network/core", branch
= "master" }
calimero-sdk-macros =  { git = "https://github.com/calimero-network/core",
branch = "master" }
borsh = { version = "1.0", features = ["derive"] }

[build-dependencies]
# Use latest release or specified version
calimero-wasm-abi = { git = "https://github.com/calimero-network/core", branch
= "master" }

[lib]
crate-type = ["cdylib"]

[dependencies.calimero-sdk]
features = ["macro"]
```

Create build.rs

> This build script automatically generates res/abi.json and res/state-schema.json from your Rust source code during cargo build by parsing src/lib.rs and extracting the ABI manifest using the calimero_wasm_abi emitter.

```
use std::fs;
use std::path::Path;

use calimero_wasm_abi::emitter::emit_manifest;

fn main() {
    println!("cargo:rerun-if-changed=src/lib.rs");

    // Parse the source code
    let src_path = Path::new("src/lib.rs");
    let src_content = fs::read_to_string(src_path).expect("Failed to read
src/lib.rs");

    // Generate ABI manifest using the emitter
    let manifest = emit_manifest(&src_content).expect("Failed to emit ABI
manifest");

    // Serialize the manifest to JSON
    let json = serde_json::to_string_pretty(&manifest).expect("Failed to
serialize manifest");

    // Write the ABI JSON to the res directory
    let res_dir = Path::new("res");
    if !res_dir.exists() {
        fs::create_dir_all(res_dir).expect("Failed to create res directory");
    }

    let abi_path = res_dir.join("abi.json");
    fs::write(&abi_path, json).expect("Failed to write ABI JSON");
```

```
    // Extract and write the state schema
    if let Ok(mut state_schema) = manifest.extract_state_schema() {
        state_schema.schema_version = "wasm-abi/1".to_owned();

        let state_schema_json =
            serde_json::to_string_pretty(&state_schema).expect("Failed to
serialize state schema");
        let state_schema_path = res_dir.join("state-schema.json");
        fs::write(&state_schema_path, state_schema_json)
            .expect("Failed to write state schema JSON");
    }
}
```

Build to WASM

```
# Add WASM target
$: rustup target add wasm32-unknown-unknown
> ... adding target ...
> or
> info: component 'rust-std' for target 'wasm32-unknown-unknown' is up to date

# Build WASM binary
$: cargo build --target wasm32-unknown-unknown --release
>    Updating git repository `https://github.com/calimero-network/core`
>    Updating crates.io index
> ...
> Finished `release` profile [optimized] target(s) in 25.67s
# Output: target/wasm32-unknown-unknown/release/my_calimero_app.wasm
```

Extract ABI

> Application ABI gets extracted automatically when building the project by using build.rs file created in previous steps.

```
$: ls res
> abi.json my-calimero-app.wasm state-schema.json
```

Best Practices

1. **Always use CRDTs**: Don't use regular Rust collections for synchronized state
2. **Use `&self` for views**: Methods with `&self` are read-only and faster (no attribute needed)
3. **Handle errors properly**: Use `app::Result<T>` and meaningful error types
4. **Use private storage for secrets**: Never put secrets in CRDT state
5. **Emit events for UI updates**: Enable real-time updates across nodes
6. **Test with multiple nodes**: Verify sync behavior in multi-node scenarios

Deep Dives

For detailed SDK documentation:

- **Application SDK**: `core/crates/sdk/README.md` - Complete API reference
- **Storage Collections**: `core/crates/storage/README.md` - CRDT types and semantics
- **Examples**: `core/apps` - Working application examples

Related Topics

- Getting Started - Complete getting started guide
- Applications - Application architecture overview
- Core Apps Examples - Reference implementations