AI VOID FIELD GUIDE

Mastering SpaceTimeDB: A Zero-to-Advanced Guide

Embark on a comprehensive journey to master SpaceTimeDB from the ground up, learning to build scalable real-time applications and multiplayer experiences with confidence.

AI VOID

2026-05-04 18:27 IST

Contents

CHAPTER 01

Chapter 1: Decoding SpaceTimeDB: Concepts and Architecture

Welcome, aspiring real-time architect, to the exciting world of SpaceTimeDB!

In this first chapter of our comprehensive guide, we're going to embark on a journey to demystify SpaceTimeDB. You'll discover what makes it a game-changer for building real-time, collaborative, and multiplayer applications. We'll explore its fundamental concepts, understand the unique architectural problems it solves, and get our hands dirty with the initial setup.

By the end of this chapter, you'll have a solid grasp of: * What SpaceTimeDB is and why it's different from traditional backend approaches. * The core architectural components that enable its magic. * How to install the SpaceTimeDB CLI and set up your first project. * A taste of the development workflow.

Ready to dive into a new paradigm for backend development? Let's go!

What is SpaceTimeDB? A Paradigm Shift

Imagine building a multiplayer game, a collaborative document editor, or a real-time dashboard. What do these applications have in common? They all need to: 1. **Store data** persistently. 2. **Execute backend logic** (e.g., game rules, access control). 3. **Synchronize state** across many connected clients in real-time.

Traditionally, you'd stitch together several technologies: a database (like PostgreSQL or MongoDB), a backend API server (Node.js, Python, Go), and a real-time layer (WebSockets, Pub/Sub services). This approach works, but it often leads to complexity, potential inconsistencies, and significant development overhead, especially when dealing with complex real-time state.

SpaceTimeDB offers a radical solution: it's a unified platform that combines a **database**, **backend logic execution**, and **real-time synchronization** into a single, cohesive system. Think of it as a shared, global state machine that all your clients connect to and interact with.

The Problems SpaceTimeDB Solves

Let's break down the common pain points that SpaceTimeDB aims to eliminate:

- **Data Consistency Across Layers:** In a traditional setup, you might write data to a database, then process it in your backend, and finally push updates to clients. Ensuring that all these layers are consistent and that clients always see the latest, correct state can be tricky, error-prone, and slow.
- **Complex Real-time Logic:** Implementing complex real-time interactions often means writing intricate WebSocket handlers, managing connection states, and manually propagating updates to relevant clients. This boilerplate can quickly become a maintenance nightmare.
- **Backend Scaling for Real-time:** Scaling a backend that handles both persistent data and high-throughput real-time updates requires careful architectural decisions and often involves separate scaling strategies for each component.
- **Deterministic State for Multiplayer:** For applications like multiplayer games, ensuring that all clients see the exact same game state at any given moment, and that actions are processed deterministically, is paramount. Traditional systems often struggle here, leading to "desync" issues.

How SpaceTimeDB's Unified Architecture Works

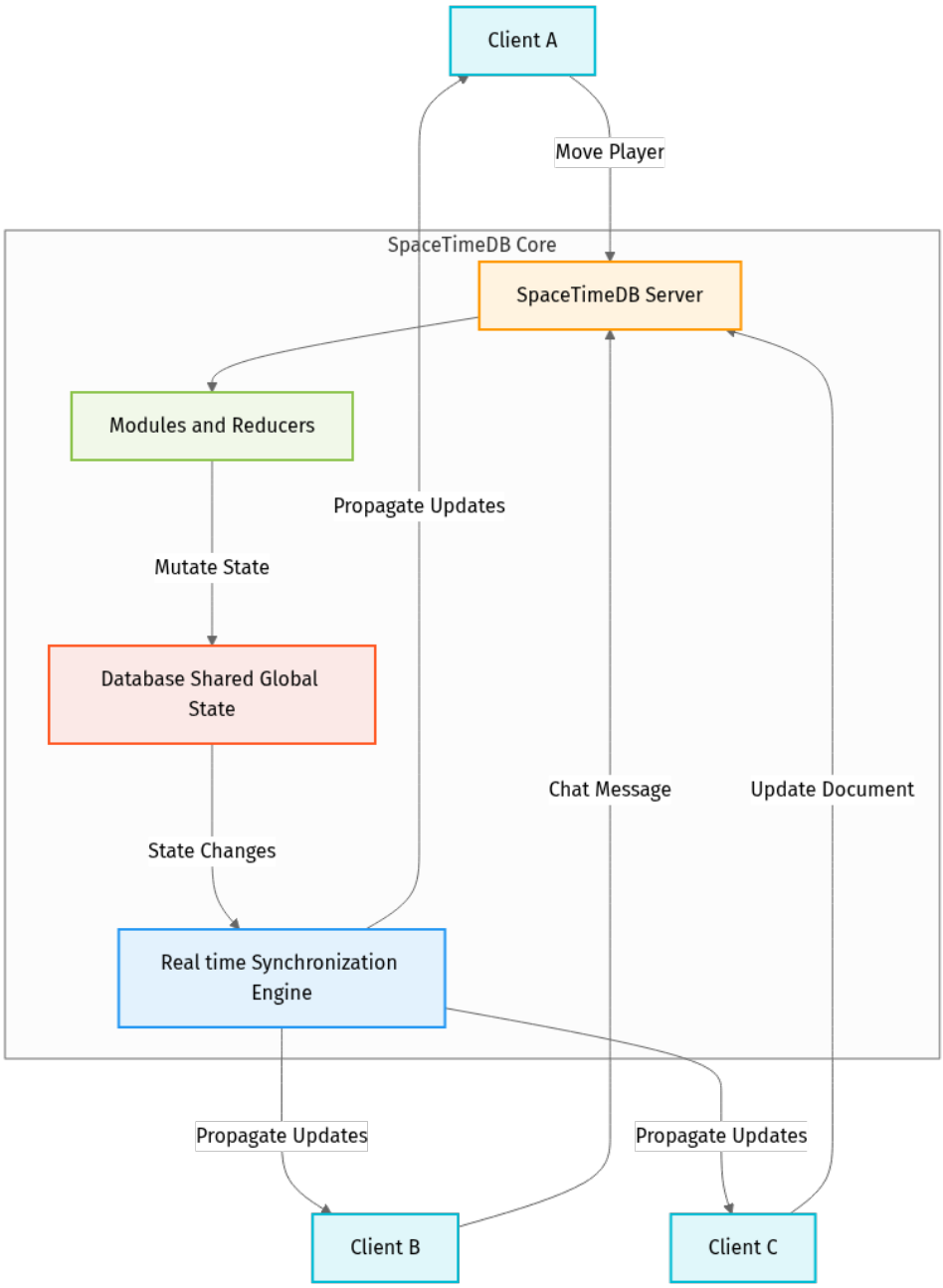
SpaceTimeDB tackles these challenges by providing a single source of truth for your application's state. Here's a conceptual overview of its architecture:

1. **Shared Global State:** At its core, SpaceTimeDB maintains a single, consistent database that represents the entire application state. This isn't just a database; it's a living, breathing model of your application.
2. **Deterministic Logic (Modules/Reducers):** Instead of separate backend API endpoints, SpaceTimeDB uses "modules" written in Rust. Within these modules, you define "reducers" – functions that explicitly describe how the application state can change. These reducers are deterministic, meaning that given the same starting state and input, they will always produce the same output state. This is crucial for consistency and debugging.
3. **Event-Driven Updates:** When a reducer modifies the state, SpaceTimeDB automatically generates "events." These events represent the changes that just occurred.

4. **Real-time Synchronization:** SpaceTimeDB automatically propagates these state changes (via events) to all connected clients that are "subscribed" to the relevant data. Clients don't poll; they receive updates as they happen, ensuring everyone is always in sync.

This unified approach simplifies development immensely. You define your data schema, write your state-changing logic (reducers), and SpaceTimeDB handles the database persistence, logic execution, and real-time synchronization for you.

Let's visualize this core architecture:



- **Client (Web/Game):** Your frontend application or game client. It sends actions to SpaceTimeDB.

- **SpaceTimeDB Server:** The central hub. It receives client actions.
- **Modules & Reducers:** This is where your application's server-side logic resides. Reducers process actions and determine how the global state should change.
- **Database:** SpaceTimeDB's internal, persistent database that stores the shared global state.
- **Real-time Synchronization Engine:** This component observes changes in the database and efficiently pushes those updates to all subscribed clients.

This architecture ensures that every client eventually sees the same, consistent state, making it incredibly powerful for building complex real-time interactions.

Setting Up Your SpaceTimeDB Development Environment

Now that we have a conceptual understanding, let's get SpaceTimeDB running on your machine!

Step 1: Install Rust and Cargo

SpaceTimeDB modules are written in Rust, a performant and safe systems programming language. You'll need the Rust toolchain installed to compile your SpaceTimeDB backend logic.

If you don't have Rust installed, the recommended way is to use `rustup`.

1. **Open your terminal or command prompt.**
2. **Run the following command:** `bash curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` This command downloads and runs the `rustup` installer. Follow the on-screen instructions. Typically, you can choose the default installation option.
3. **After installation, you might need to restart your terminal or source your shell's profile** (e.g., `source $HOME/.cargo/env`) to ensure `cargo` (Rust's package manager and build tool) is in your PATH.
4. **Verify your Rust installation:** `bash rustc --version cargo --version` You should see output showing the installed versions of `rustc` and `cargo`. As of 2026-03-14, you'll likely see a version like `rustc 1.80.0` or similar.

Step 2: Install the SpaceTimeDB CLI

The SpaceTimeDB Command Line Interface (CLI) is your primary tool for interacting with SpaceTimeDB, including creating projects, running development servers, and deploying.

1. **Open your terminal.**
2. **Install the CLI using `cargo`:** `bash cargo install spacetimedb_cli` This command compiles and installs the `spacetimedb_cli` executable. This might take a few moments as Rust compiles the tool.
3. **Verify the SpaceTimeDB CLI installation:** `bash spacetime --version` You should see the installed version. Based on the latest stable releases, we'll assume `v2.1.0` is the current stable CLI version as of 2026-03-14. `# Expected output (approximate) spacetime_cli 2.1.0` Great! You now have the SpaceTimeDB CLI ready to roll.

Step 3: Create Your First SpaceTimeDB Project

Let's use the CLI to initialize a new SpaceTimeDB project. This will set up a basic structure for your backend module.

1. **Navigate to a directory where you want to create your project:** `bash cd ~/projects # Or any directory you prefer`
2. **Create a new SpaceTimeDB project:** `bash spacetime new my_first_spacetime_app`
    - `spacetime new`: The command to create a new project.
    - `my_first_spacetime_app`: The name of your new project directory.

    The CLI will create a new directory with the specified name and populate it with a basic SpaceTimeDB module structure.
3. **Explore the project structure:** `bash cd my_first_spacetime_app ls -F` You'll see something like this: `Cargo.toml src/ Spacetime.toml`
    - `Cargo.toml`: The manifest file for your Rust project, defining dependencies and metadata.
    - `src/`: This directory will contain your Rust source code for SpaceTimeDB modules.
    - `Spacetime.toml`: This is the SpaceTimeDB-specific configuration file for your project.

Step 4: Run the Development Server

Now, let's fire up your SpaceTimeDB development server! This server will compile your Rust modules, manage your database, and provide the real-time synchronization.

1. **Make sure you are inside your project directory (`my_first_spacetime_app`).** `bash pwd # Should show /path/to/my_first_spacetime_app`
2. **Start the development server:** `bash spacetime dev` You'll see a lot of output as SpaceTimeDB compiles your initial module and starts the server. Look for messages indicating that the server is running and listening for connections, typically on `ws://127.0.0.1:9000`.

   ```

   # Example output snippet (might vary slightly)

   ... [INFO] SpacetimeDB server listening on 127.0.0.1:9000 [INFO] SpacetimeDB server running in development mode. ... ``` Congratulations! Your first SpaceTimeDB server is up and running. It's now waiting for clients to connect and interact with its shared global state.

   To stop the server, simply press `Ctrl+C` in your terminal.

---

## Mini-Challenge: Your First Project Warm-up

It's your turn to practice!

**Challenge:** 1. Create a second SpaceTimeDB project with a different name (e.g., `my_second_app`). 2. Navigate into its directory. 3. Start its development server. 4. Observe the output and confirm it's running on the default port.

**Hint:** Remember the `spacetime new <project-name>` command and `spacetime dev`.

**What to Observe/Learn:** * You'll notice that `spacetime dev` takes some time on the first run for compilation. Subsequent runs are often faster. * The output clearly states the WebSocket address where the server is listening. This is where your clients will connect. * Each `spacetime dev` instance runs its own isolated database and module.
   ```

Common Pitfalls & Troubleshooting

Even with clear instructions, things can sometimes go sideways. Here are a few common issues and how to tackle them:

- **"command not found: rustc" or "command not found: cargo"**:
- **Reason:** Rust and Cargo are not correctly installed or their binaries are not in your system's PATH.
- **Solution:** Rerun the `rustup` installation script. Ensure you restart your terminal after installation, or manually source `~/.cargo/env` (`source $HOME/.cargo/env`).
- **"command not found: spacetime"**:
- **Reason:** The `spacetimedb_cli` was not installed correctly via `cargo install`, or `cargo`'s binary directory is not in your PATH.
- **Solution:** Double-check the `cargo install spacetimedb_cli` command. Ensure `~/.cargo/bin` is in your PATH (usually set up by `rustup`).
- **"Address already in use" or similar port error when running `spacetime dev`**:
- **Reason:** Another process (perhaps a previous `spacetime dev` instance you forgot to close, or another application) is already using port 9000.
- **Solution:** 1. Find and terminate the process using port 9000. On Linux/macOS, you can use `lsof -i :9000` to find the PID and then `kill <PID>`. On Windows, `netstat -ano | findstr :9000` then `taskkill /PID <PID> /F`. 2. Alternatively, you can specify a different port for `spacetime dev` using a command-line flag (we'll cover more advanced CLI usage later). For now, resolving the port conflict is the easiest.

Summary

Phew! You've just taken your first significant steps into the SpaceTimeDB ecosystem. Let's recap what we've covered:

- **SpaceTimeDB is a unified platform** that merges database, backend logic, and real-time synchronization, simplifying the creation of complex real-time applications.

- It solves problems like **data inconsistency**, **complex real-time logic**, and **backend scaling challenges** by providing a single, consistent source of truth and deterministic logic.
- Its core architecture revolves around a **shared global state**, **deterministic modules/reducers**, and an **event-driven real-time synchronization engine**.
- You successfully **installed Rust and the SpaceTimeDB CLI (`v2.1.0`)**.
- You **created your first SpaceTimeDB project** and **ran its development server**, seeing it listen for connections on `ws://127.0.0.1:9000`.

You've laid the groundwork! In the next chapter, we'll dive deeper into how we actually define that shared global state using SpaceTimeDB's schema, and how we start interacting with it using tables and basic queries. Get ready to start modeling your application's world!

References

- SpaceTimeDB Official Documentation
- SpaceTimeDB GitHub Repository
- Rust Programming Language

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

Chapter 10: Optimizing Performance: Indexing, Query Tuning, and Data Structures

Introduction: Making Your Real-Time Apps Fly

Welcome back, intrepid SpaceTimeDB adventurer! In our previous chapters, we've explored the foundational elements of SpaceTimeDB: setting up your environment, designing schemas, writing reducers, and synchronizing real-time state with clients. You've learned how to build a reactive, collaborative backend with ease.

But what happens when your application grows? When thousands, or even millions, of players or users are interacting with your system simultaneously? That's when performance becomes not just a nice-to-have, but a critical requirement. Slow queries, inefficient data access, or poorly designed schemas can quickly turn a blazing-fast real-time experience into a frustrating lag-fest.

In this chapter, we're going to roll up our sleeves and dive into the exciting world of performance optimization within SpaceTimeDB. We'll uncover how to leverage indexing to dramatically speed up your data lookups, explore strategies for tuning your queries, and discuss how intelligent data structure design can lay the groundwork for a truly scalable application. Our goal is for you to understand the underlying principles so you can confidently build high-performance, real-time systems that delight your users.

Ready to make your SpaceTimeDB application fly? Let's get started!

Core Concepts: The Pillars of Performance

Optimizing performance in any database system, including SpaceTimeDB, revolves around making data access as efficient as possible. This means reducing the amount of work the database has to do to find, filter, or sort the data you need.

SpaceTimeDB's Execution Model: A Quick Recap

Before we talk about optimization, let's briefly recall how SpaceTimeDB works. Your application logic lives in **reducers**, which are deterministic functions that

modify the shared, global state stored in **tables**. When a reducer executes, it might need to read data from various tables. The speed at which it can access this data directly impacts the overall performance of your system, affecting both reducer execution time and the real-time propagation of changes to clients.

The Power of Indexing: Your Data's Table of Contents

Imagine you have a massive textbook with hundreds of pages, and you need to find every mention of "quantum physics." You could read the entire book cover-to-cover, but that would take ages! Or, you could flip to the index at the back, find "quantum physics," and it would tell you exactly which pages to turn to.

Database indexes work in much the same way. An **index** is a special lookup table that the database engine can use to speed up data retrieval. It stores a sorted copy of the data from one or more columns in a table, along with pointers to the full rows.

Why are indexes important?

- **Faster Reads:** Indexes dramatically accelerate queries that involve filtering (`WHERE` clauses), sorting (`ORDER BY` clauses), and joining data.
- **Unique Constraints:** Indexes are also used to enforce uniqueness constraints (e.g., ensuring no two users have the same username).

What's the catch?

- **Storage Overhead:** Indexes take up disk space.
- **Write Performance Impact:** Every time you `insert`, `update`, or `delete` data in an indexed column, the index itself must also be updated. This adds overhead and can slow down write operations.

The key is to strike a balance: add indexes where they provide significant read benefits, but avoid over-indexing, which can hurt write performance without providing proportional read gains.

Creating Indexes in SpaceTimeDB

SpaceTimeDB, being built with Rust, leverages Rust's attribute system to define your schema and indexes. You'll add special attributes to your table struct fields to declare them as primary keys, unique, or general indexes.

Here are the primary attributes for indexing:

- `#[primary_key]`: This attribute designates a field as the table's primary key.

- **Purpose:** Uniquely identifies each row in the table.
- **Behavior:** SpaceTimeDB automatically creates a highly optimized, unique index for the primary key. Lookups using the primary key are extremely fast. A table **must** have a primary key. You can also define a compound primary key using a tuple.
- `#[unique]`: This attribute ensures that the values in this field are unique across all rows in the table.
- **Purpose:** Enforce data integrity (e.g., unique usernames, email addresses).
- **Behavior:** SpaceTimeDB creates a unique index for this field, allowing for fast lookups and ensuring no duplicate values are inserted.
- `#[index]`: This is a general-purpose secondary index.
- **Purpose:** Speed up queries that filter or sort by this field, or fields within a tuple if it's a compound index.
- **Behavior:** SpaceTimeDB creates a non-unique index. You can have multiple rows with the same value in an `#[index]` field.

Let's look at an example to solidify this.

```
// In your `stdb_modules/src/lib.rs` file
use spacetimedb::{spacetimedb, table, ReducerContext, Identity, Timestamp};

// Define your table struct
#[spacetimedb(table)]
#[derive(Clone, Debug)]
pub struct UserProfile {
    #[primary_key] // This field is the unique identifier for each user
    pub user_id: u32,

    #[unique] // Ensures no two users have the same username
    pub username: String,

    #[index] // Index this for fast lookups when searching by email
    pub email: String,

    pub display_name: String,
    pub last_login: u64,
}
```

Explanation: * `user_id` is our `primary_key`, allowing for lightning-fast lookups by ID. * `username` is marked `unique`, ensuring every user has a distinct username and providing a fast lookup path. * `email` is marked `index`, which is useful if we frequently search for users by their email address, but don't need it to be unique (e.g., if multiple accounts could share an email in some specific scenarios, though `unique` is often preferred here too).

Query Tuning Strategies

Indexes are powerful, but they are only effective if your queries are written to utilize them. SpaceTimeDB's client SDKs (like TypeScript) provide methods that intelligently interact with the underlying indexed data.

1. **Prefer `find` over `filter` when possible:**
    - `find_by_primary_key(key)`: This method (or its equivalent in other languages) is the fastest way to retrieve a single row because it directly uses the primary key index.
    - `find_by_unique_field(value)`: Similarly, if you have a `#[unique]` field, SpaceTimeDB generates a `find_by_unique_field` method, which also uses an optimized index lookup.
    - `filter()...`: The `filter` method is more general and allows for complex conditions. While it can use secondary indexes, it might be slower than direct `find` operations if a primary or unique key lookup is possible.
2. **Order of `filter` clauses:** While SpaceTimeDB's query planner is smart, placing the most restrictive `filter` clauses first can sometimes help, especially if they align with an index. For example, if you have an index on `game_id` and `score`, filtering by `game_id` first will narrow down the dataset efficiently.
3. **Minimize data transfer (client-side):** While SpaceTimeDB's real-time synchronization often means clients subscribe to entire tables or filtered views, being mindful of the data you actually need can inform your schema design. If a client only ever needs a `user_id` and `username` for a leaderboard, perhaps a separate, denormalized `LeaderboardEntry` table containing just those fields could be more efficient than subscribing to full `UserProfile` objects for every player.

Optimizing Data Structures (Schema Design)

The way you structure your data in tables has a profound impact on performance. This goes beyond just adding indexes; it's about how you model your entities and their relationships.

1. **Normalization vs. Denormalization:**

- **Normalization:** Breaking down data into smaller, related tables to reduce redundancy and improve data integrity. Example: Separate `Users` and `UserScores` tables. Good for complex relationships and ensuring data

consistency. Can lead to more "joins" (conceptual joins in SpaceTimeDB, as you'd query multiple tables).

- **Denormalization:** Intentionally adding redundant data to tables to improve read performance. Example: Storing a player's `username` directly in the `PlayerScore` table, even though `username` also exists in `UserProfile`. This avoids an extra lookup when displaying a leaderboard.
- **SpaceTimeDB Context:** Due to its real-time nature, denormalization can be particularly attractive. If you denormalize data, and the source data changes, you'd typically need a reducer to update the denormalized copy. However, with SpaceTimeDB's reactive model, clients will instantly see the updated denormalized data. The trade-off is often more complex write logic in reducers versus simpler, faster reads for clients. For read-heavy, real-time scenarios like leaderboards or dashboards, a controlled amount of denormalization is often beneficial.

1. **Choosing Appropriate Data Types:**
    - Use the smallest data type that can accurately represent your data. `u32` for IDs if they won't exceed 4 billion, instead of `u64`.
    - Avoid excessively large strings or binary data if not strictly necessary.
    - SpaceTimeDB's native types are optimized. Stick to them where possible.
2. **Embedding Data for Frequent Access:** If certain pieces of related data are almost always accessed together, consider embedding them directly into a single table row. For example, instead of a `GameSession` table and a `GameSettings` table, you might embed the `GameSettings` struct directly into the `GameSession` table's row if settings are small and rarely change independently. This avoids needing to query two separate tables.

Internal Mechanics of Query Execution (Brief)

When you define indexes and perform queries, SpaceTimeDB's internal engine does a lot of heavy lifting. It maintains efficient B-tree-like data structures for each index. When a query comes in (either from a reducer or a client subscription), SpaceTimeDB's query planner analyzes the conditions (`filter` clauses, `order_by` clauses) and selects the most efficient index (or combination of indexes) to fulfill the request. If no suitable index is found, it will resort to a full table scan, which is much slower for large datasets. This process is largely automatic, so your job is to provide the right indexes for your common query patterns.

Step-by-Step Implementation: Building an Optimized Leaderboard

Let's put these concepts into practice with a common real-time scenario: a game leaderboard. We'll start with a basic schema and incrementally add indexes, explaining the benefits at each step.

Scenario: A Simple Game Leaderboard

We want to store player scores for various games. Our goal is to quickly: 1. Find a player's current score. 2. Get the top players globally. 3. Get the top players for a specific game.

1. Initial Schema: No Specific Indexes (Yet)

Let's begin with a `PlayerScore` table without any explicit secondary indexes.

Open your `stdb_modules/src/lib.rs` file and add the following table definition:

```
// stdb_modules/src/lib.rs
use spacetimedb::{spacetimedb, table, ReducerContext, Identity, Timestamp};

#[spacetimedb(table)]
#[derive(Clone, Debug)]
pub struct PlayerScore {
    // We'll use a compound primary key for player_id and game_id
    // This ensures a player can only have one score per game, and provides
fast lookups
    // for a specific player's score in a specific game.
    #[primary_key]
    pub player_id_game_id: (u32,
u32), // Tuple (player_id, game_id) as primary key

    pub player_id: u32,
    pub game_id: u32,
    pub score: u32,
    pub timestamp: u64, // When the score was last updated
}
```

Explanation: * We've defined a `PlayerScore` table. * The `player_id_game_id` field is a **compound primary key**, a tuple of `(player_id, game_id)`. This is a powerful feature of SpaceTimeDB, ensuring that each player has only one score entry per game. It also means that finding a player's score for a specific game will be incredibly fast. * The individual `player_id` and `game_id` fields are still present for convenience and future indexing. * At this stage, we can efficiently find a specific player's score for a specific game using the primary key. But what about finding all scores for a player, or the top scores globally?

2. Adding Secondary Indexes for Global Leaderboards and Player History

Now, let's enhance our table with secondary indexes to support more complex query patterns efficiently.

Modify your `PlayerScore` struct in `stdb_modules/src/lib.rs` to include `#[index]` attributes:

```
// stdb_modules/src/lib.rs
use spacetimedb::{spacetimedb, table, ReducerContext, Identity, Timestamp};

#[spacetimedb(table)]
#[derive(Clone, Debug)]
pub struct PlayerScore {
    #[primary_key]
    pub player_id_game_id: (u32, u32),

    #[index] // Index for finding all scores for a specific player
    pub player_id: u32,

    #[index] // Index for finding all scores for a specific game
    pub game_id: u32,

    #[index] // Index for efficiently sorting by score (e.g., global top
scores)
    pub score: u32,

    pub timestamp: u64,
}
```

Explanation: * `#[index] pub player_id: u32`: This index allows us to quickly retrieve all scores for a given `player_id`, regardless of the game. Useful for a "My Scores" section. * `#[index] pub game_id: u32`: This index enables fast filtering for all scores belonging to a particular `game_id`. Essential for per-game leaderboards. * `#[index] pub score: u32`: This index is crucial for sorting. If we want to find the top 10 scores globally, SpaceTimeDB can use this index to efficiently sort and retrieve the highest scores without scanning the entire table.

3. Reducer Example: Updating Scores

Let's quickly add a reducer that can update player scores. This reducer will benefit from the primary key index.

Add this reducer to the same `stdb_modules/src/lib.rs` file:

```rust
// stdb_modules/src/lib.rs (after your table definition)

#[spacetimedb(reducer)]
pub fn submit_score(ctx: ReducerContext, player_id: u32, game_id: u32, new_scor
e: u32) -> Result<(), String> {
    let pk = (player_id, game_id); // Construct the primary key tuple

    match table::PlayerScore::find_by_player_id_game_id(&pk) { // Uses the
compound primary_key index
        Some(mut player_score) => {
            // Only update if the new score is higher
            if new_score > player_score.score {
                player_score.score = new_score;
                player_score.timestamp = ctx.timestamp.as_micros();
                player_score.update(); // Update the existing row
            }
        },
        None => {
            // New entry for this player and game
            table::PlayerScore::insert(PlayerScore {
                player_id_game_id: pk,
                player_id,
                game_id,
                score: new_score,
                timestamp: ctx.timestamp.as_micros(),
            })?;
        }
    }
    Ok(())
}
```

Explanation: * The `submit_score` reducer handles score submissions. * It constructs the compound primary key `(player_id, game_id)`. * It then uses `table::PlayerScore::find_by_player_id_game_id(&pk)` to efficiently check if an entry for this player and game already exists. This lookup is incredibly fast because it's using the primary key index. * If an entry exists, it updates the score if the `new_score` is higher. Otherwise, it inserts a new entry.

4. Client-Side Querying (Conceptual)

While we're focusing on the Rust module, it's important to understand how client-side queries (e.g., from a TypeScript frontend) would benefit from these indexes.

Example Client-Side Queries (TypeScript):

- **Get a specific player's score for a specific game:** `typescript // This would internally use the primary key index const score = PlayerScore.findById([playerId, gameId]);` - **Get top 10 global scores:** `typescript // This query would benefit from the `score` index for sorting const topScores = PlayerScore.filter() .orderBy("score", "desc") .take(10) .all();` - **Get top 5 scores for a specific game**

(e.g., `gameId = 123`): `typescript // This benefits from both `game_id` index for filtering and `score` index for sorting const topGameScores = PlayerScore.filter(score => score.game_id === 123) .orderBy("score", "desc") .take(5) .all();`

Notice how the `filter` and `orderBy` operations directly map to the indexes we've created. SpaceTimeDB's client SDKs are designed to translate these high-level queries into efficient operations that leverage the server-side indexes.

Mini-Challenge: Optimizing for Recent Activity

You've done a great job setting up your leaderboard! Now, let's add another common real-time requirement.

Challenge: Your game developers want to add a new "Recent High Scores" section. This section should display the top 3 highest scores submitted in the last 24 hours.

1. **Identify the missing index:** What additional index (or modification to an existing one) would be most beneficial to make this specific query pattern performant?
2. **Modify the `PlayerScore` schema:** Add the necessary attribute to your `PlayerScore` struct in `stdb_modules/src/lib.rs`.
3. **Conceptual Client Query:** Write down (or mentally formulate) how a client-side query (e.g., in TypeScript) would look to retrieve these "Recent High Scores," leveraging your new index.

Hint: Think about what fields are involved in filtering by "last 24 hours" and sorting by "highest scores." How can you combine these efficiently?

What to Observe/Learn: This challenge highlights how anticipating common query patterns directly informs your indexing strategy. A single index might not cover all combinations of filters and sorts, sometimes requiring compound indexes or multiple single-column indexes.

Common Pitfalls & Troubleshooting

Even with a good understanding of indexing, it's easy to fall into common traps.

1. **Over-indexing:** While indexes speed up reads, every index adds overhead to write operations (inserts, updates, deletes) and consumes storage. Too

many indexes, especially on tables with high write traffic, can actually slow down your application overall.

- **Troubleshooting:** Regularly review your indexes. Are all of them actively used by critical queries? Remove unused indexes. 2. **Under-indexing:** This is the most common performance culprit. If a frequently executed query filters or sorts on a column without an index, SpaceTimeDB will have to perform a full table scan, which is very slow for large tables.
- **Troubleshooting:** Use profiling tools (if available in SpaceTimeDB's ecosystem, or general application profiling) to identify slow queries. If a query consistently takes too long and involves filtering/sorting on a particular column, consider adding an index. 3. **Not understanding query patterns:** Performance problems often arise because indexes were chosen based on assumptions, not actual query analytics. The "top N scores" query might be frequent, but "scores for a specific player on a specific date range" might be rare.
- **Troubleshooting:** Monitor your application's usage. What are the most common read patterns? Design your indexes to optimize these critical paths. 4. **Inefficient Reducer Logic:** Indexes optimize data access, but they can't fix inefficient logic within your reducers. If your reducer performs complex, unoptimized computations, or iterates over large datasets unnecessarily after retrieving them, performance will still suffer.
- **Troubleshooting:** Profile your reducer execution times. Can any computations be pre-calculated or simplified? Can you retrieve only the necessary data?

Summary: Key Takeaways for Performance

Congratulations! You've navigated the complexities of performance optimization in SpaceTimeDB. Here's a quick recap of the vital concepts:

- **Indexing is paramount for read performance:** It drastically speeds up filtering, sorting, and lookups by providing efficient data access paths.
- **SpaceTimeDB uses Rust attributes for indexing:** `#[primary_key]`, `#[unique]`, and `#[index]` are your tools for defining indexed fields.
- **Balance read and write costs:** Indexes improve reads but add overhead to writes and consume storage. Choose indexes wisely.

- **Design your schema for your query patterns:** Anticipate how your application will access data and create indexes that support those common operations.
- **Consider denormalization strategically:** For read-heavy, real-time scenarios, controlled denormalization can simplify client-side queries and boost performance, especially with SpaceTimeDB's instant state propagation.
- **SpaceTimeDB's client SDKs leverage indexes:** Your high-level client queries (e.g., `filter().orderBy()`) automatically benefit from well-defined server-side indexes.

By applying these principles, you're well-equipped to design and build SpaceTimeDB applications that are not just real-time, but also incredibly fast and scalable.

What's Next?

In the next chapter, we'll delve deeper into advanced topics like **Concurrency Handling and Transactions**. You'll learn how SpaceTimeDB ensures data consistency and integrity even in highly concurrent environments, and how you can leverage its transactional model for robust application logic.

References

- SpaceTimeDB Official Documentation
- SpaceTimeDB GitHub Repository
- Rust `#[attribute]` documentation (general concept)
- Database Indexing Explained (General Concepts)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Chapter 11: Scaling Your SpaceTimeDB Application: Distributed Architectures and Deployment

Chapter 11: Scaling Your SpaceTimeDB Application: Distributed Architectures and Deployment

Welcome back, intrepid SpaceTimeDB adventurer! Up until now, we've focused on building fantastic real-time applications on a single SpaceTimeDB instance. But what happens when your game explodes in popularity, your collaborative app goes viral, or your real-time dashboard needs to handle millions of data points per second? That's when you need to think about scaling.

In this chapter, we're going to tackle one of the most exciting and critical aspects of building production-ready systems: making them scale. We'll explore how SpaceTimeDB's unique architecture lends itself to distributed deployments, dive into concepts like sharding and replication, and then discuss modern deployment strategies using tools like Docker and Kubernetes. Get ready to design systems that can handle immense loads and stay resilient!

Before we dive in, ensure you're comfortable with SpaceTimeDB's core concepts, including reducers, tables, client synchronization, and event-driven updates, as covered in previous chapters. This chapter builds on that foundation, applying those concepts to a distributed environment.

Understanding Scaling: Why and How

Scaling is all about ensuring your application can handle increased demand without sacrificing performance, availability, or reliability. Imagine a single lane road trying to handle rush hour traffic; it quickly becomes congested. To solve this, you either widen the road (vertical scaling) or add more roads (horizontal scaling).

- **Vertical Scaling (Scaling Up):** This means adding more resources (CPU, RAM, faster disk) to a single server. It's often the easiest first step, but it hits physical limits quickly and creates a single point of failure. You can only make a server so big!

- **Horizontal Scaling (Scaling Out):** This involves adding more servers or instances to distribute the load. This is generally preferred for modern, highly available, and performant applications because it offers near-limitless potential and resilience. If one server fails, others can pick up the slack.

SpaceTimeDB is inherently designed for horizontal scaling. Its deterministic, event-sourced nature makes it an excellent candidate for distributed architectures where multiple instances work together seamlessly.

SpaceTimeDB's Scaling Philosophy

SpaceTimeDB's core strength for scaling lies in its deterministic execution model. Recall that all state changes in SpaceTimeDB happen via `reducers`. When a reducer is invoked, it's guaranteed to produce the same output state given the same input state and arguments, regardless of which SpaceTimeDB node executes it. This is a game-changer for distributed systems!

In a distributed SpaceTimeDB cluster, multiple nodes can cooperate. They maintain a consistent, shared state by agreeing on the sequence of events (reducer invocations). Even if a client connects to a different node, or if a node goes down and comes back up, the system can ensure a consistent view of the database.

This philosophy allows SpaceTimeDB to:

1. **Maintain Strong Consistency:** Despite being distributed, SpaceTimeDB aims for strong consistency, meaning all clients see the same, most up-to-date state.
2. **Achieve High Availability:** If one node fails, others can continue serving requests, minimizing downtime.
3. **Distribute Workload:** Incoming client connections and reducer invocations can be spread across multiple nodes.

Distributed Architecture Patterns for SpaceTimeDB

When we talk about horizontal scaling for databases, two key patterns emerge: **sharding** and **replication**.

1. Sharding (Data Partitioning)

Imagine your database as a giant library. If it gets too big for one building, you might split it into several smaller libraries, each holding a different collection of books (e.g., "Fiction," "Science," "History"). This is sharding!

What it is: Sharding involves partitioning your data across multiple independent database instances, called shards. Each shard holds a subset of the total data.
Why it's important:

- **Performance:** Queries only need to hit a smaller dataset on a specific shard, leading to faster response times.
- **Storage Limits:** A single server has finite storage. Sharding allows you to store virtually unlimited data across many servers.
- **Write Throughput:** Writes are distributed across multiple shards, increasing the overall write capacity of the system.

How SpaceTimeDB can leverage Sharding: With SpaceTimeDB, you might decide to shard your data based on a specific key, like a `game_id`, `user_id`, or `organization_id`. All data related to a specific game, user, or organization would reside on a single shard.

For example, in a multiplayer game: * `Shard A` might handle all game rooms with `game_id` ending in 0-3. * `Shard B` might handle all game rooms with `game_id` ending in 4-7. * `Shard C` might handle all game rooms with `game_id` ending in 8-9 and A-F (hexadecimal).

When a client wants to join `game_id: "abcde123"`, your application logic (or a proxy layer) would determine which shard (`Shard A` in this case) holds that game's data and route the client to a SpaceTimeDB node serving that shard.

2. Replication (High Availability and Read Scaling)

Now, let's say your "Fiction" library is super popular, and everyone wants to read the same books. To handle the demand, you might create several identical copies of the "Fiction" library. This is replication!

What it is: Replication involves creating multiple copies of the same data (or shard) across different SpaceTimeDB instances. **Why it's important:**

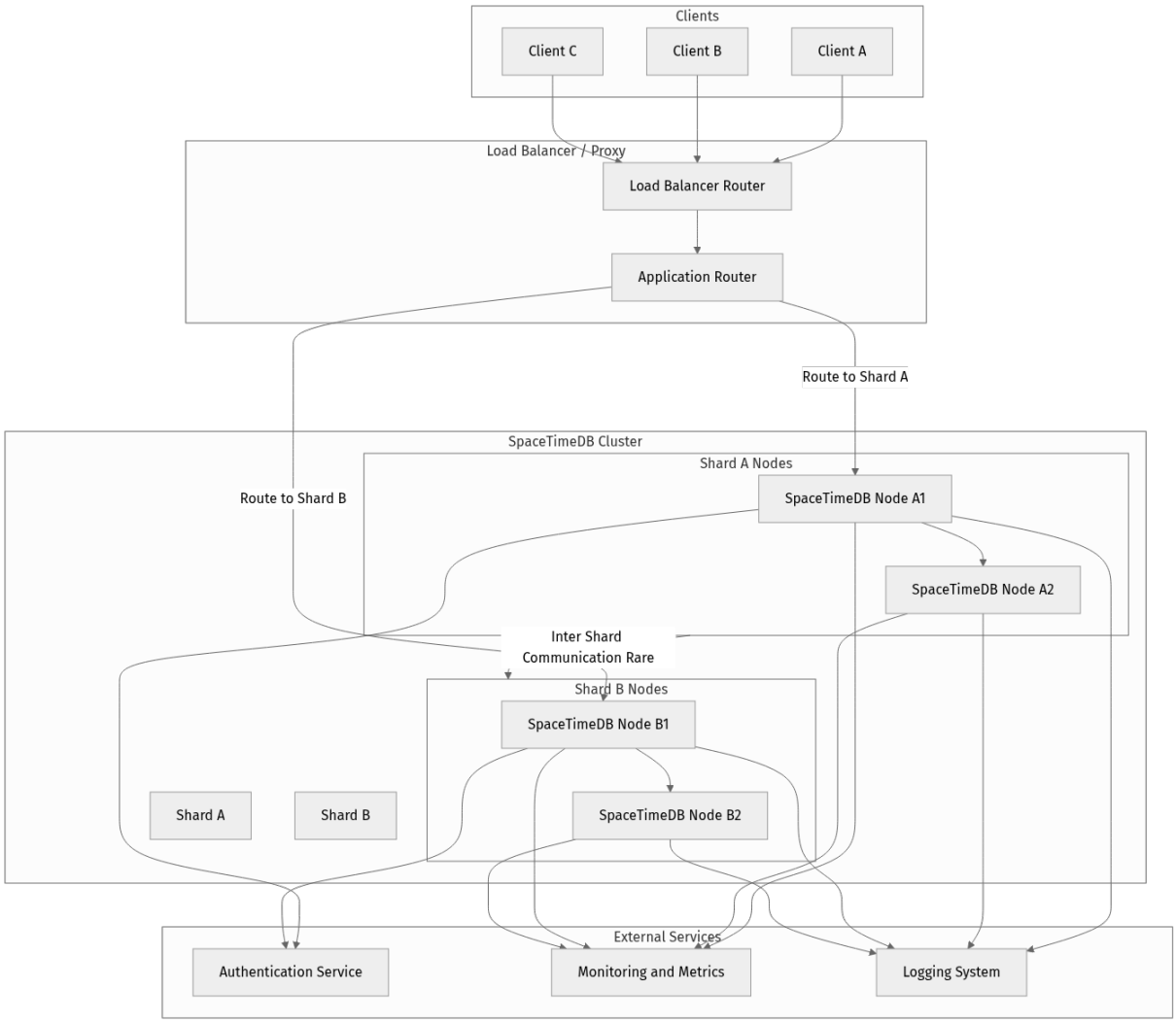
- **High Availability:** If one node fails, a replica can immediately take over, ensuring continuous service. This is critical for systems that cannot tolerate downtime.
- **Read Scaling:** Clients can read data from any replica. By distributing read requests across multiple replicas, you can significantly increase your system's read throughput.
- **Disaster Recovery:** Replicas can be placed in different geographical regions or data centers, protecting against regional outages.

How SpaceTimeDB can leverage Replication: Each shard in a sharded setup can have its own replicas. For instance, `Shard A` might have `Node A1` (primary) and `Node A2` (replica). If `Node A1` goes offline, `Node A2` can become the new primary, and clients can reconnect to it.

SpaceTimeDB's deterministic nature makes replication straightforward: all replicas execute the same sequence of events, ensuring they maintain an identical copy of the shard's state.

Distributed Architecture Diagram

Let's visualize how these components fit together:



Explanation of the Diagram:

- **Clients:** Your frontend applications or game clients. They don't directly know about shards or replicas.
- **Load Balancer / Proxy:** This layer is crucial. It distributes incoming client connections across your SpaceTimeDB cluster. It might also contain your "Application Router" logic.

- **Application Router:** This is custom logic (often part of your API Gateway or a dedicated service) that determines which SpaceTimeDB shard a client should connect to based on the data they need to access (e.g., a specific `user_id` or `game_id`).
- **SpaceTimeDB Cluster:** This is where the magic happens!
- **Shards:** The data is partitioned into `Shard A` and `Shard B`. Each shard handles a distinct subset of your application's data.
- **Replication:** Each shard is replicated across multiple nodes (e.g., `Node A1` and `Node A2` for `Shard A`). If `Node A1` fails, `Node A2` can take over.
- **Inter-Node Communication:** SpaceTimeDB nodes within a shard (primary and replicas) communicate to ensure state consistency. Inter-shard communication is generally discouraged for performance reasons, as it complicates the sharding logic.

Deployment Strategies: From Local to Cloud

Now that we understand the architecture, how do we actually deploy such a system? Modern deployments heavily rely on containerization and orchestration.

1. Containerization with Docker

What it is: Docker allows you to package your SpaceTimeDB application (including its dependencies and configuration) into a lightweight, portable container. This ensures that your application runs consistently across different environments, from your local machine to production servers.

Why it's important:

- **Consistency:** "Works on my machine" becomes "works everywhere."
- **Isolation:** Containers isolate your application from the host system and other applications.
- **Portability:** Move containers easily between development, testing, and production.
- **Efficiency:** Containers are much lighter than traditional virtual machines.

SpaceTimeDB and Docker: The SpaceTimeDB CLI (version `2.x` as of 2026-03-14) can generate a `Spacetime.toml` configuration file and your module. You can then use a `Dockerfile` to build an image containing your SpaceTimeDB module and the `spacetimedb` server executable.

Let's assume the official SpaceTimeDB Docker image is available (e.g., `clockworklabs/spacetimedb-server:2.x`).

2. Orchestration with Kubernetes

What it is: Kubernetes (K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It's like a conductor for your Docker containers.

Why it's important for SpaceTimeDB:

- **Automated Deployment:** Define your desired state (e.g., "run 3 SpaceTimeDB nodes for Shard A"), and Kubernetes handles getting there.
- **Self-Healing:** If a SpaceTimeDB node crashes, Kubernetes automatically restarts it or replaces it.
- **Scaling:** Easily scale up or down the number of SpaceTimeDB nodes based on demand.
- **Load Balancing:** Kubernetes provides built-in load balancing to distribute traffic to your SpaceTimeDB instances.
- **Service Discovery:** Nodes can find each other automatically.

SpaceTimeDB and Kubernetes: You would define Kubernetes manifests (YAML files) for:

- **Deployments:** To manage your SpaceTimeDB nodes (e.g., `Deployment` for Shard A, `Deployment` for Shard B).
- **StatefulSets:** For databases like SpaceTimeDB that require stable, unique network identifiers and persistent storage.
- **Services:** To expose your SpaceTimeDB nodes to other services or the outside world.
- **ConfigMaps / Secrets:** To manage configuration files (like `Spacetime.toml`) and sensitive data.

Networking Considerations for Distributed Deployments

In a distributed setup, networking is paramount. Here are key points:

- **Low Latency:** For real-time applications, the network latency between your clients and the SpaceTimeDB cluster, and especially between SpaceTimeDB nodes themselves, is critical. Deploying nodes in the same region/availability zone is often a good start.
- **High Bandwidth:** SpaceTimeDB constantly synchronizes state and propagates events. Ensure your network can handle the data throughput.
- **Firewall Rules:** Carefully configure firewalls to allow:
    - Client connections to your load balancer/proxy.

    - Load balancer/proxy connections to your SpaceTimeDB nodes.
    - Inter-node communication between SpaceTimeDB instances (e.g., for replication or consensus protocols).

- **DNS & Service Discovery:** In Kubernetes, `Services` and `StatefulSets` provide robust service discovery, allowing your SpaceTimeDB nodes to find each other by name rather than hardcoded IPs.

Observability for Scaled Systems

When you have many SpaceTimeDB nodes running, it becomes impossible to manually check each one. Observability tools are your eyes and ears:

- **Logging:** Centralize logs from all SpaceTimeDB instances. Use structured logging (e.g., JSON format) for easier analysis. Tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Grafana Loki are popular.
- **Metrics:** Collect key performance indicators (KPIs) from each SpaceTimeDB node. These might include:
    - CPU and Memory Usage
    - Network I/O
    - Client Connection Count
    - Reducer Execution Latency
    - Event Throughput
    - Database Query Latency
    - Replication Lag Prometheus and Grafana are standard tools for this.
- **Tracing:** For complex interactions across multiple services and SpaceTimeDB nodes, distributed tracing (e.g., OpenTelemetry, Jaeger) helps you understand the flow of requests and identify bottlenecks.

Step-by-Step Implementation: Simulating a Multi-Node SpaceTimeDB with Docker Compose

While a full Kubernetes deployment is complex, we can simulate a multi-node SpaceTimeDB setup locally using Docker Compose to understand the concepts. We'll set up two SpaceTimeDB nodes, each running the same module, and configure them to potentially interact.

Prerequisites: * Docker Desktop installed (includes Docker Compose). * SpaceTimeDB CLI (`v2.x`). * A SpaceTimeDB project created (e.g., from Chapter 2), let's call it `my_multiplayer_game`.

1. Prepare Your SpaceTimeDB Module

Navigate into your SpaceTimeDB project directory. Ensure your module compiles correctly.

```
cd my_multiplayer_game
spacetime generate # If you made schema changes
spacetime build
```

This will create `target/spacetime_module.wasm`.

2. Create a `Dockerfile` for Your Module

We'll create a simple `Dockerfile` to package your compiled module with the SpaceTimeDB server. Create a file named `Dockerfile` in your `my_multiplayer_game` project root:

```
# Use a minimal base image, e.g., Alpine Linux
FROM alpine:3.19

# Install necessary runtime dependencies (if any)
# For SpaceTimeDB, often just a basic environment is enough

# Set environment variables for SpaceTimeDB
ENV SPDB_BIND_ADDRESS="0.0.0.0:3000"
ENV SPDB_DB_PATH="/data/db"
ENV SPDB_MODULE_PATH="/app/spacetime_module.wasm"
ENV SPDB_NODE_NAME="spacetime-node" # This will be overridden by docker-compose

# Create necessary directories
RUN mkdir -p /app /data/db

# Copy the compiled SpaceTimeDB module
COPY target/spacetime_module.wasm /app/spacetime_module.wasm

# Expose the default SpaceTimeDB port
EXPOSE 3000

# Command to run the SpaceTimeDB server
ENTRYPOINT ["/usr/local/bin/spacetimedb"]
CMD ["--module", "/app/spacetime_module.wasm", "--bind-address",
"0.0.0.0:3000", "--db-path", "/data/db"]
```

Important Note: The `ENTRYPOINT` and `CMD` here assume the `spacetimedb` executable is available. In a real scenario, you'd either build `spacetimedb` from source into this image, or more commonly, use an official `clockworklabs/spacetimedb-server` base image and then just copy your `wasm` module into it. For simplicity, we're assuming `spacetimedb` is magically there in this Alpine image for demonstration purposes, or you could add a step to download it.

A more robust `Dockerfile` would look like this, using an official base image:

```
FROM clockworklabs/spacetimedb-server:2.0.0-rc.3 # Use latest stable 2.x
version

# Set environment variables
ENV SPDB_BIND_ADDRESS="0.0.0.0:3000"
ENV SPDB_DB_PATH="/data/db"
ENV SPDB_MODULE_PATH="/app/spacetime_module.wasm"
ENV SPDB_NODE_NAME="spacetime-node"

# Create necessary directories
RUN mkdir -p /app /data/db

# Copy the compiled SpaceTimeDB module from your project
# Assuming this Dockerfile is in your project root and `target/
spacetime_module.wasm` exists
COPY target/spacetime_module.wasm /app/spacetime_module.wasm

# Expose the default SpaceTimeDB port
EXPOSE 3000

# The base image already has the ENTRYPOINT set to spacetimedb,
# so we just need to provide the arguments for CMD.
CMD ["--module", "/app/spacetime_module.wasm", "--bind-address",
"0.0.0.0:3000", "--db-path", "/data/db"]
```

For this example, we'll use the second, more realistic `Dockerfile` that leverages the official base image. Make sure to check SpacetimeDB GitHub Releases for the absolute latest stable `2.x` release tag for the `FROM` line. As of 2026-03-14, let's assume `2.0.0-rc.3` is the latest stable release candidate or full release.

3. Create a `docker-compose.yml` File

Create a file named `docker-compose.yml` in your `my_multiplayer_game` project root:

```yaml
version: '3.8'

services:
  spacetime-node-1:
    build: . # Build from the Dockerfile in the current directory
    container_name: spacetime-node-1
    ports:
      - "3000:3000" # Expose port 3000 on host for node 1
    environment:
      SPDB_NODE_NAME: "node-1"
      SPDB_DB_PATH: "/data/db/node-1" # Unique path for persistent data
    volumes:
      - node1_data:/data/db/node-1 # Persistent volume for node 1's data
    networks:
      - spacetimedb_network

  spacetime-node-2:
    build: .
    container_name: spacetime-node-2
    ports:
      - "3001:3000" # Expose node 2 on host port 3001
    environment:
      SPDB_NODE_NAME: "node-2"
      SPDB_DB_PATH: "/data/db/node-2"
    volumes:
      - node2_data:/data/db/node-2
    networks:
      - spacetimedb_network

networks:
  spacetimedb_network:
    driver: bridge

volumes:
  node1_data:
  node2_data:
```

Explanation of `docker-compose.yml`:

- **`services`**: Defines the two SpaceTimeDB nodes (`spacetime-node-1` and `spacetime-node-2`).
- **`build: .`**: Tells Docker Compose to build the image for this service using the `Dockerfile` in the current directory.
- **`container_name`**: Assigns a friendly name to each container.
- **`ports`**: Maps container port `3000` to different host ports (`3000` for node 1, `3001` for node 2) so you can access them individually from your machine.
- **`environment`**: Sets environment variables for each container. `SPDB_NODE_NAME` is important for identification. `SPDB_DB_PATH` is unique for each node to prevent data conflicts.

- **`volumes`**: Uses Docker volumes (`node1_data`, `node2_data`) to persist the database state for each node. This means if you stop and restart the containers, your data will still be there.
- **`networks`**: Creates a custom bridge network (`spacetimedb_network`) so the nodes can communicate with each other internally using their service names (`spacetime-node-1`, `spacetime-node-2`).

4. Run Your Multi-Node SpaceTimeDB Cluster

In your project directory, open a terminal and run:

```
docker compose up --build -d
```

- `docker compose up`: Starts the services defined in `docker-compose.yml`.
- `--build`: Ensures your Docker images are rebuilt if there are changes to your `Dockerfile` or context.
- `-d`: Runs the containers in detached mode (in the background).

You should see output indicating the containers are being created and started.

To check if they are running:

```
docker ps
```

You should see `spacetime-node-1` and `spacetime-node-2` listed.

5. Connect Clients to Individual Nodes

Now you have two SpaceTimeDB nodes running! You can connect your SpaceTimeDB clients (e.g., from your frontend application) to either `ws://localhost:3000` (for node 1) or `ws://localhost:3001` (for node 2).

For a truly distributed setup, you would typically put a load balancer in front of these nodes, and clients would connect to the load balancer's address. The load balancer would then distribute connections to the available SpaceTimeDB nodes.

Connecting with the CLI (for testing):

You can use the SpaceTimeDB CLI to connect and inspect each node:

```
spacetime client connect ws://localhost:3000
# Inside the client:
# > status
# > get table_name
# Press Ctrl+C to disconnect

spacetime client connect ws://localhost:3001
# Inside the client:
# > status
# > get table_name
# Press Ctrl+C to disconnect
```

Observe that even though they are separate nodes, if they were configured for replication or sharding (which would involve more advanced `Spacetime.toml` configuration than shown here), they would reflect a consistent state. For this basic Docker Compose, they are currently independent instances running the same module. To truly make them a cluster, SpaceTimeDB's internal clustering features (which might involve peer discovery and consensus configuration in `Spacetime.toml`) would need to be enabled.

Mini-Challenge: Add a Third Node and Implement a Basic Load Balancer (Conceptual)

Your turn to get hands-on!

Challenge: 1. Modify the `docker-compose.yml` file to add a third SpaceTimeDB node, `spacetime-node-3`. 2. Expose `spacetime-node-3` on host port `3002`. 3. Ensure it uses its own persistent volume and node name. 4. (Conceptual) Imagine how you would implement a simple client-side load balancer that randomly picks one of the three node addresses (`ws://localhost:3000`, `ws://localhost:3001`, `ws://localhost:3002`) for a new client connection. You don't need to write the client code, just describe the logic.

Hint: * You can copy and paste the `spacetime-node-2` service block, then adjust the `container_name`, `ports`, `environment`, and `volumes` for `spacetime-node-3`. * For the client-side load balancer, think about how you might store a list of available endpoints and select one from that list.

What to Observe/Learn: * How easily Docker Compose allows you to scale out your application locally. * The importance of unique configurations (ports, volumes, names) for each node. * The foundational concept of distributing client connections across multiple instances.

Common Pitfalls & Troubleshooting

1. **Network Configuration Issues:**

- **Symptom:** Clients can't connect, or nodes can't communicate.
- **Troubleshooting:** * Check `docker ps` to ensure containers are running and port mappings are correct. * Verify firewall rules on your host machine. * Inside Docker Compose, containers on the same network can communicate by service name (e.g., `spacetime-node-1` can talk to `spacetime-node-2` at `spacetime-node-2:3000`). * Ensure `SPDB_BIND_ADDRESS` is set to `0.0.0.0` inside the container to listen on all interfaces.

1. **Resource Starvation:**

- **Symptom:** Slow performance, nodes crashing under load.
- **Troubleshooting:** * Monitor CPU, memory, and disk I/O of your containers and host. * Use `docker stats` to get real-time resource usage for containers. * Ensure your server hardware (or cloud instance type) is sufficient for the workload. * Consider optimizing your SpaceTimeDB module (reducers, queries) for efficiency.

1. **Incorrect Sharding Key or Data Distribution:**

- **Symptom:** One shard becomes a "hot spot" (much higher load than others), leading to performance bottlenecks.
- **Troubleshooting:** * Analyze your data access patterns. * Choose a sharding key that evenly distributes your data and workload across shards. * Periodically review shard utilization metrics to identify imbalances.

1. **Lack of Observability:**

- **Symptom:** You don't know why your distributed system is slow or failing.
- **Troubleshooting:** * Implement centralized logging. * Set up comprehensive metrics collection and dashboards. * Use tracing for complex request flows. * Without these, debugging a distributed system is like flying blind!

Summary

Phew! You've just taken a massive leap into the world of advanced SpaceTimeDB deployment. Here are the key takeaways from this chapter:

- **Scaling is Essential:** For real-world applications, understanding how to scale is crucial for handling growth and maintaining reliability.

- **Horizontal Scaling is Key:** Adding more instances is generally preferred over bigger instances for SpaceTimeDB, thanks to its deterministic, event-sourced nature.
- **Sharding Partitions Data:** Distributes your database across multiple SpaceTimeDB instances, improving performance and storage capacity for large datasets.
- **Replication Ensures Availability:** Creates copies of your data for fault tolerance and read scaling.
- **Modern Deployment Relies on Containers & Orchestration:** Docker provides consistent environments, and Kubernetes automates the management, scaling, and self-healing of your distributed SpaceTimeDB cluster.
- **Networking is Critical:** Pay close attention to latency, bandwidth, and firewall rules in distributed setups.
- **Observability is Non-Negotiable:** Centralized logging, metrics, and tracing are your best friends for understanding and troubleshooting complex distributed systems.

You've learned how SpaceTimeDB's design philosophy makes it a powerful platform for building scalable, real-time applications. While setting up a production-grade distributed system involves many more considerations (like advanced consensus protocols, dynamic scaling, and robust CI/CD pipelines), this chapter has given you the foundational knowledge and a practical taste of how to begin.

What's Next?

In the next chapter, we'll delve into the crucial topic of **Security Models and Authentication Integration** to ensure your scalable SpaceTimeDB applications are not only performant and available but also protected from unauthorized access and malicious activity.

References

1. **SpacetimeDB Official Documentation:** The primary source for all SpacetimeDB features, architecture, and CLI usage. https://spacetimedb.com/docs

2. **SpacetimeDB GitHub Repository:** For the latest releases, source code, and community contributions. https://github.com/clockworklabs/SpacetimeDB
3. **Docker Official Documentation:** Comprehensive guides for containerization. https://docs.docker.com/
4. **Kubernetes Official Documentation:** Extensive resources for container orchestration. https://kubernetes.io/docs/
5. **CNCF Observability Overview:** Introduction to logging, metrics, and tracing in cloud-native environments. https://www.cncf.io/blog/2021/04/29/an-introduction-to-observability/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Chapter 12: Security & Authentication in SpaceTimeDB

Introduction to Security & Authentication in SpaceTimeDB

Welcome to Chapter 12! As we venture further into building sophisticated real-time applications with SpaceTimeDB, securing our data and controlling access becomes paramount. Just as you wouldn't leave your front door unlocked, we can't deploy an application without robust authentication and authorization mechanisms. This chapter will equip you with the knowledge and practical skills to safeguard your SpaceTimeDB applications.

In this chapter, we'll unravel SpaceTimeDB's unique approach to security, which tightly integrates authentication and authorization directly into your backend logic (reducers). We'll explore how to identify users, manage their identities, and critically, how to enforce granular permissions for every action and data access within your SpaceTimeDB instance. By the end, you'll be able to design and implement secure, multi-user real-time systems with confidence.

Before diving in, we assume you're comfortable with SpaceTimeDB's core concepts, including defining schemas, writing reducers, and understanding the client-server interaction, as covered in previous chapters. Let's make our real-time applications not just fast and collaborative, but secure!

Core Concepts: Protecting Your Real-Time World

Securing real-time, collaborative applications presents unique challenges. Unlike traditional request-response systems where a single HTTP request might carry all necessary authentication headers, real-time systems maintain persistent connections, and state changes can originate from multiple clients simultaneously. SpaceTimeDB addresses this by embedding security directly into its deterministic, event-driven model.

The Security Landscape: Authentication vs. Authorization

Let's start by clarifying two fundamental terms that are often used interchangeably but have distinct meanings:

- **Authentication:** This is the process of verifying who a user is. When you log in with a username and password, or use a social login (like Google or GitHub), you are authenticating. The system confirms your identity.
- **Authorization:** This is the process of determining what an authenticated user is allowed to do. Once the system knows who you are, it decides if you have permission to read a specific piece of data, write to a particular table, or execute a certain action.

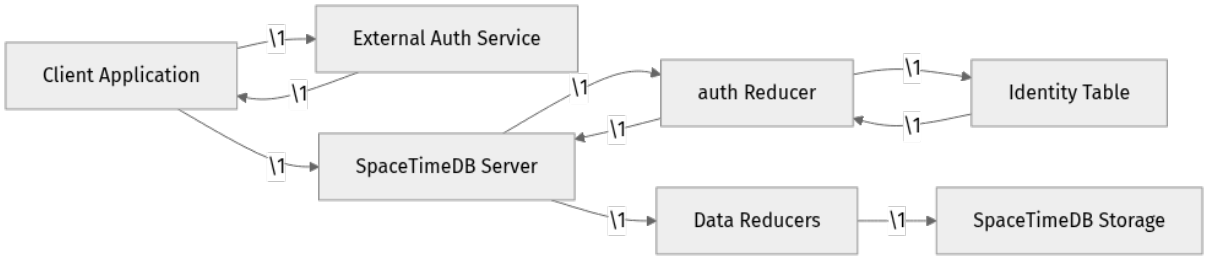
SpaceTimeDB handles both, but with a clear separation of concerns. You'll typically integrate with an external authentication provider and then use SpaceTimeDB's built-in mechanisms for authorization.

SpaceTimeDB's Security Model Overview

SpaceTimeDB's security model centers around a few key ideas:

1. **The `Caller` Object:** Every reducer call in SpaceTimeDB has access to a `caller` object. This object holds information about the client that initiated the reducer call, most importantly, their `identity_id`.
2. **The `auth` Reducer:** This special reducer is responsible for authenticating clients. When a client connects to SpaceTimeDB and wants to be identified, it calls the `auth` reducer with some form of credentials (e.g., a JWT, an API key). The `auth` reducer validates these credentials and returns an `IdentityId`.
3. **The `identity` Table:** This is a special, internal SpaceTimeDB table that stores unique identifiers for authenticated clients. The `auth` reducer manages entries in this table.
4. **Reducer-based Authorization:** Once a client is authenticated and has an `identity_id` associated with their connection (via the `caller` object), all subsequent data-modifying reducers can inspect `caller.identity_id` to enforce authorization rules.

Let's visualize this flow:



Explanation of the Flow:

1. **Client Authenticates Externally:** Your frontend application first authenticates the user with an external identity provider (like Auth0, Firebase Auth, Google, etc.). This is crucial because SpaceTimeDB itself is not an identity provider.
2. **Token Received:** The external service returns a secure token (commonly a JSON Web Token or JWT) to the client.
3. **Client Connects to SpaceTimeDB:** The client connects to your SpaceTimeDB instance, typically passing this token as part of the connection or immediately calling the `auth` reducer.
4. **`auth` Reducer Invocation:** The client explicitly calls your custom `auth` reducer, passing the token.
5. **Token Validation & Identity Management:** Inside the `auth` reducer, you validate the token (e.g., verify its signature, expiration, issuer). If valid, you either retrieve an existing identity from the `identity` table or create a new one.
6. **IdentityId Returned:** The `auth` reducer returns the `IdentityId` associated with the authenticated user.
7. **Server Associates Identity:** SpaceTimeDB internally associates this `IdentityId` with the client's connection.
8. **`caller` Object Populated:** For all subsequent reducer calls made by this client, the `caller` object will contain `caller.identity_id`, allowing your reducers to know who is making the request.
9. **Authorization in Data Reducers:** Your application's data reducers (e.g., `create_post`, `update_profile`) can then check `caller.identity_id` to enforce specific authorization rules before modifying or querying data in `DBStorage`.

The `grant_access` Statement

While most authorization is handled by logic within your reducers, SpaceTimeDB also provides a `grant_access` statement. This is a powerful feature for defining

table-level or row-level read permissions directly in your schema. It specifies which `IdentityId`s (or roles/groups derived from identities) are allowed to subscribe to or query specific data. This is particularly useful for preventing unauthorized clients from even seeing data they shouldn't have access to, reducing network traffic and simplifying reducer logic.

For example, you might `grant_access` to a `private_messages` table only to the identities that are participants in the conversation, or to an `admin_dashboard` table only to identities with an "admin" role. We'll focus on reducer-based authorization first for actions, and then touch upon `grant_access` for read permissions.

Step-by-Step Implementation: Building a Simple Authenticated System

Let's put these concepts into practice. We'll build a simplified authentication and authorization system. For the sake of this tutorial, we'll mock external token validation. In a real application, you would integrate with an external library or service to verify JWTs.

First, let's ensure you have SpaceTimeDB CLI `v2.x` installed. If not, refer to Chapter 2 for installation instructions.

1. Initialize Your Project

If you don't have a project already, let's create a new one:

```
# Ensure you have the SpaceTimeDB CLI (v2.x)
# Install via: curl -sSL https://get.spacetimedb.com | bash
# Verify: spacetime --version

# Create a new project directory
mkdir my_secure_app
cd my_secure_app

# Initialize SpaceTimeDB project
spacetime new --name my_secure_app
```

2. Define the `User` Table

While SpaceTimeDB automatically manages the `identity` table, we'll often want to store additional user-specific data (e.g., username, email, profile picture). Let's define a `User` table that links directly to an `IdentityId`.

Open `src/lib.spacetimedb/schema.spacetimedb` and add the following:

```
// src/lib.spacetimedb/schema.spacetimedb

// The Identity table is automatically managed by SpaceTimeDB.
// We'll define a User table that links to an Identity.

// @spacetimedb_id: This attribute marks 'identity_id' as the primary key
// and also indicates it's linked to an Identity.
// It ensures a one-to-one relationship between a User and an Identity.
table User {
    #[spacetimedb(primarykey)]
    #[spacetimedb(id)]
    identity_id: IdentityId, // Links directly to a SpaceTimeDB Identity
    username: String,
    email: String,
    created_at: u64,
}

// A simple Post table that will be associated with a User/Identity
table Post {
    #[spacetimedb(primarykey)]
    post_id: u64,
    author_identity_id: IdentityId, // Who wrote the post
    content: String,
    created_at: u64,
}
```

After modifying the schema, always run:

```
spacetime generate
```

This command updates the generated Rust code for your reducers and types.

3. Implement the `auth` Reducer

Now, let's create the `auth` reducer. This reducer will take a "token" (a simple string in our mocked example), validate it, and return an `IdentityId`.

Open `src/lib.spacetimedb/mod.rs` and add the following reducer logic. Remember, in a real application, `validate_token_and_extract_user_info` would involve cryptographic verification of a JWT with a public key from your external auth provider.

```rust
// src/lib.spacetimedb/mod.rs

use spacetimedb::{
    spacetimedb,
    Identity, IdentityId,
    // Add other necessary imports, e.g., for `User` if you're using it
    // use crate::User; // Assuming User is in the same module or imported
};

// Import the generated schema types
use crate::{
    User, Post,
    // Add other generated types as needed
};

// --- Helper for Mock Token Validation (DO NOT USE IN PRODUCTION) ---
// In a real application, this would involve verifying a JWT signature,
// checking claims, expiration, etc., using a robust JWT library.
fn validate_token_and_extract_user_info(token: String) -> Option<(String, Strin
g)> {
    // For this example, let's mock a simple token validation.
    // A real token would be a complex JWT.
    if token.starts_with("mock_token_") {
        let parts: Vec<&str> = token.split('_').collect();
        if parts.len() == 3 {
            let username = parts[1].to_string();
            let email = format!("{}@example.com", parts[1]);
            return Some((username, email));
        }
    }
    None
}
// --- End Helper ---

/// The `auth` reducer is special. It's called by clients to authenticate
themselves.
/// It receives a `token` (e.g., a JWT) and, if valid, returns an `IdentityId`.
/// SpaceTimeDB then associates this `IdentityId` with the client's connection.
#[spacetimedb(reducer)]
pub fn auth(token: String) -> IdentityId {
    // 1. Validate the token and extract user information.
    // In a real app, you'd use a JWT library (e.g., `jsonwebtoken` crate in
Rust)
    // to verify the token's signature, expiry, and claims.
    let user_info_option = validate_token_and_extract_user_info(token.clone());

    let (username, email) = match user_info_option {
        Some(info) => info,
        None => {
            // If the token is invalid, we can either:
            // a) Return a default, unauthenticated IdentityId
(IdentityId::default() or 0)
            // b) Panic, which will disconnect the client (more secure for
critical apps)
            // For now, let's panic for clear error handling during
development.
            panic!("Invalid or malformed authentication token: {}", token);
        }
    };

    // 2. Get or create an Identity.
```

```
    // SpaceTimeDB automatically creates an `Identity` entry if one doesn't
exist
    // for the given token/user info. The `Identity` table is managed
internally.
    // We typically use a unique identifier from the external auth provider
    // to map to a SpaceTimeDB Identity. For our mock, we'll use the username.
    let identity_id = Identity::filter_by_identity_id(IdentityId::hash_from_byt
es(username.as_bytes()))
        .map(|i| i.identity_id)
        .unwrap_or_else(|| {
            // If no existing identity, create a new one.
            // SpaceTimeDB's `Identity` table is special and often managed
implicitly
            // or through specific API calls. For direct interaction, we often
            // derive a stable IdentityId from a unique external identifier.
            // Here, we're using a hash of the username as a stable ID.
            // In a real system, you'd use the user ID from your external auth
provider.
            let new_identity_id =
IdentityId::hash_from_bytes(username.as_bytes());
            // Note: We don't explicitly `insert` into `Identity` here.
            // SpaceTimeDB handles the creation of the `Identity` entry when an
`IdentityId`
            // is returned by the `auth` reducer or used with `grant_access`.
            new_identity_id
        });


    // 3. Create or update the `User` record associated with this Identity.
    // We use `IdentityId::hash_from_bytes(username.as_bytes())` to ensure a
consistent ID.
    if !User::filter_by_identity_id(identity_id).exists() {
        // If the user doesn't exist, create a new User record.
        User::insert(User {
            identity_id,
            username: username.clone(),
            email: email.clone(),
            created_at: spacetimedb::timestamp(),
        }).unwrap(); // Handle potential errors in a production app
    } else {
        // Optionally update existing user data, e.g., if email changes in
external system.
        // For simplicity, we'll just ensure it exists.
        // let mut user = User::filter_by_identity_id(identity_id).unwrap();
        // user.username = username;
        // user.email = email;
        // user.update().unwrap();
    }

    // 4. Return the IdentityId. This is crucial!
    // SpaceTimeDB will associate this IdentityId with the client's connection.
    identity_id
}
```

Let's break down the `auth` reducer:

- `validate_token_and_extract_user_info`: This is a helper function that simulates validating an external authentication token. In a production environment, this would involve much more robust logic, typically using a

JWT library to verify the token's signature against a public key, checking its expiration, and extracting claims like `sub` (subject/user ID) and `email`.

- `IdentityId::hash_from_bytes(username.as_bytes())`: We're deriving a stable `IdentityId` from the username. In a real system, you'd use the immutable user ID provided by your external authentication service (e.g., `sub` claim from a JWT) to ensure consistency. This ID is how SpaceTimeDB internally tracks the authenticated client.
- `User::insert` or `User::filter_by_identity_id`: After successfully authenticating, we either create a new `User` entry in our custom `User` table or ensure an existing one is up-to-date. This links our application-specific user data to the SpaceTimeDB `IdentityId`.
- `return identity_id`: The critical step! By returning an `IdentityId`, SpaceTimeDB knows who this client is for the duration of their connection. All subsequent reducer calls from this client will have access to this `identity_id` via the `caller` object.

4. Implement an Authenticated Reducer (`create_post`)

Now that we have an `auth` reducer, let's create a reducer that requires authentication and uses the `caller` object for authorization. We'll make a `create_post` reducer that only allows authenticated users to create posts, and automatically associates the post with the author's identity.

Add this to `src/lib.spacetimedb/mod.rs`:

```
// src/lib.spacetimedb/mod.rs (continued)

// Ensure `spacetimedb::timestamp()` is available for `created_at`
use spacetimedb::timestamp;

/// Reducer to create a new post, requiring an authenticated caller.
#[spacetimedb(reducer)]
pub fn create_post(caller: Identity, content: String) {
    // Authorization step: Ensure the caller is authenticated.
    // If caller.identity_id is IdentityId::default() or 0, it means the client
    // has not successfully authenticated via the `auth` reducer.
    if caller.identity_id == IdentityId::default() {
        panic!("Unauthorized: You must be logged in to create a post.");
    }

    // Optional: Check if the identity has a corresponding User entry
    // This adds another layer of validation, ensuring a 'complete' user.
    if !User::filter_by_identity_id(caller.identity_id).exists() {
        panic!("Unauthorized: No user profile found for this identity.");
    }

    // Generate a unique post_id. In a real app, you might use a UUID or
    // a more robust sequence generator. For simplicity, we'll hash the content
    // and timestamp.
    let post_id = spacetimedb::hash_bytes(&format!("{}-{}-{}", caller.identity_
id, content, timestamp()).as_bytes());

    // Insert the new post, associating it with the author's identity.
    Post::insert(Post {
        post_id,
        author_identity_id: caller.identity_id,
        content,
        created_at: timestamp(),
    }).unwrap();

    spacetimedb::log!("Post created successfully by identity: {:?}", caller.ide
ntity_id);
}
```

Key Points in `create_post`:

- **`caller: Identity`**: This is how SpaceTimeDB injects the `caller` object into your reducer. It contains the `identity_id` of the client making the call.
- **`if caller.identity_id == IdentityId::default()`**: This is your authorization check. If `caller.identity_id` is the default (unauthenticated) ID, we `panic!`, preventing the action. This is a simple but effective authorization gate.
- **`author_identity_id: caller.identity_id`**: We automatically associate the post with the authenticated user's identity, preventing spoofing.

5. Start SpaceTimeDB and Test

Now, let's run SpaceTimeDB and test our authentication and authorization.

```
spacetime dev
```

This will start your SpaceTimeDB instance. You can interact with it using the CLI or a client SDK.

Testing with `spacetime call` (CLI):

1. **Try to create a post unauthenticated:**

   `bash spacetime call create_post 'Hello, unauthenticated world!'`

   You should see an error message similar to: `Reducer call failed: "Unauthorized: You must be logged in to create a post."` This confirms our authorization check works!

2. **Authenticate as a user:**

   `bash spacetime call auth 'mock_token_alice'`

   This should succeed and return an `IdentityId`. SpaceTimeDB's CLI maintains the authenticated session for subsequent calls within the same CLI instance.

3. **Try to create a post authenticated as Alice:**

   `bash spacetime call create_post 'Hello from Alice!'`

   This should succeed! You'll see a log message in your `spacetime dev` console.

4. **Authenticate as a different user:**

   `bash spacetime call auth 'mock_token_bob'`

   Now the CLI is authenticated as Bob.

5. **Create a post as Bob:**

   `bash spacetime call create_post 'Bob was here.'`

   This will also succeed, and the post will be associated with Bob's identity.

You can inspect the `User` and `Post` tables in the SpaceTimeDB admin UI (usually `http://localhost:3000`) to see the created entries.

Mini-Challenge: Update Your Own Profile

Now it's your turn to apply what you've learned!

Challenge: Create a new reducer called `update_profile` that allows an authenticated user to update only their own username and email in the `User` table.

Hints:

1. The reducer should take `caller: Identity`, a `new_username: String`, and `new_email: String` as arguments.
2. Perform an authorization check at the beginning, similar to `create_post`.
3. Use `User::filter_by_identity_id(caller.identity_id).unwrap()` to find the current user's profile.
4. Ensure you are only updating the `User` entry whose `identity_id` matches `caller.identity_id`. If you tried to update another user's profile, it should panic or return an error.

What to Observe/Learn: This challenge reinforces how to use `caller.identity_id` for **row-level authorization**, ensuring users can only modify data they own or are authorized to change.

Common Pitfalls & Troubleshooting

1. **Forgetting to check `caller.identity_id`:** The most common mistake! If you don't explicitly check `caller.identity_id` in your data-modifying reducers, any client (even unauthenticated ones) could potentially call them and modify your data. Always assume reducers are publicly callable unless secured.
2. **Incorrect Token Validation:** In a real application, if your `auth` reducer's token validation logic is flawed (e.g., not verifying signatures, not checking expiration, or accepting invalid issuers), you risk authenticating malicious clients. Use battle-tested libraries for JWT validation.
3. **Confusing `IdentityId::default()` with a valid identity:** `IdentityId::default()` (which is `0`) represents an unauthenticated caller. Never treat it as a valid, authenticated user ID.
4. **Overly Complex Authorization Logic:** While reducers are powerful, try to keep authorization logic concise and clear. For very complex role-based access control (RBAC) or attribute-based access control (ABAC), you might pre-process roles/permissions in your `auth` reducer or an external service and store them in the `User` table, then check these simpler flags in your data reducers.

5. **Not running `spacetime generate`:** Any changes to `schema.spacetimedb` require `spacetime generate` to update the Rust types, otherwise your `mod.rs` might have compilation errors.
6. **Client-side `auth` reducer call:** Remember that the client must explicitly call the `auth` reducer with their token. SpaceTimeDB does not automatically infer identity from a connection string.

Summary

Phew! We've covered a lot of ground today, laying the foundation for secure SpaceTimeDB applications. Here are the key takeaways:

- **Authentication vs. Authorization:** Authentication verifies who you are, authorization determines what you can do.
- **SpaceTimeDB's Security Core:** Relies on the `caller` object, the special `auth` reducer, and the internal `identity` table.
- **External Authentication:** SpaceTimeDB integrates with external identity providers (Auth0, Firebase, etc.) for authenticating users and issuing tokens.
- **The `auth` Reducer's Role:** It validates external tokens, manages SpaceTimeDB `Identity` entries (often by deriving a stable `IdentityId` from an external user ID), and returns the `IdentityId` to be associated with the client's connection.
- **Reducer-based Authorization:** Your data-modifying reducers use the `caller.identity_id` to enforce granular access control, ensuring only authorized users can perform specific actions or modify specific data.
- **The `grant_access` Statement:** A powerful schema-level tool for defining read permissions, preventing unauthorized clients from even subscribing to sensitive data.

You now have the tools to build real-time applications that are not just highly collaborative but also robustly secure. In the next chapters, we'll continue to explore advanced topics, including deployment and scaling strategies, to bring your SpaceTimeDB applications to production readiness!

References

- SpaceTimeDB Official Documentation - Authentication
- SpaceTimeDB Official Documentation - Reducers

- SpaceTimeDB Official Documentation - Schema
- JSON Web Token (JWT) Official Website
- Auth0 Documentation - What is Authentication?

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

Chapter 13: Project: Building a Real-time Collaborative Whiteboard

Chapter 13: Project: Building a Real-time Collaborative Whiteboard

Welcome back, intrepid SpaceTimeDB explorer! In this chapter, we're going to put many of the concepts you've learned into practice by building a truly exciting project: a real-time collaborative whiteboard. Imagine multiple users drawing simultaneously on the same canvas, seeing each other's strokes appear instantly – that's the magic we'll create with SpaceTimeDB.

This project will solidify your understanding of how SpaceTimeDB excels at managing dynamic, shared state for interactive applications. We'll design a schema for drawing data, implement reducers to handle drawing actions, and conceptualize the client-side integration that brings it all to life. You'll see firsthand how SpaceTimeDB's built-in real-time synchronization makes building such complex features surprisingly straightforward.

Before we dive in, make sure you're comfortable with: * SpaceTimeDB CLI setup and project initialization (Chapter 2) * Defining database schemas with tables and fields (Chapter 4) * Writing server-side logic using reducers (Chapter 6) * Connecting client applications and subscribing to table changes (Chapter 7)

Ready to sketch out some awesome real-time collaboration? Let's get started!

Core Concepts for a Collaborative Whiteboard

Building a collaborative whiteboard requires a few key pieces of information to be managed and synchronized in real time: the strokes themselves, and potentially information about who is currently active on the board.

1. Modeling Drawing Strokes

How do we represent a drawing on a whiteboard? A drawing is typically composed of multiple "strokes." Each stroke usually has: * A unique identifier. * The user who created it. * A color. * A thickness (or brush size). * A series of points that define its path.

SpaceTimeDB tables are perfect for storing this kind of structured data. We can create a `Stroke` table where each row represents a single continuous line drawn by a user. The series of points can be stored as a JSON array or a similar structured type within a field.

2. User Presence (Optional but Recommended)

For a truly collaborative experience, it's often helpful to know who else is currently viewing or drawing on the whiteboard. This can be modeled with a `UserPresence` table, tracking which users are active on which board. While we might not fully implement cursors in this chapter, having a presence table sets the stage for such features.

3. Real-time Synchronization in Action

The real power of SpaceTimeDB for this project is its automatic real-time synchronization. When one user draws a stroke, their client will call a SpaceTimeDB reducer. This reducer updates the `Stroke` table. Immediately, SpaceTimeDB detects this change and propagates the new stroke data to all other connected clients subscribed to the `Stroke` table. This happens without you writing any explicit WebSocket or pub/sub code – SpaceTimeDB handles it all!

Let's visualize this flow:

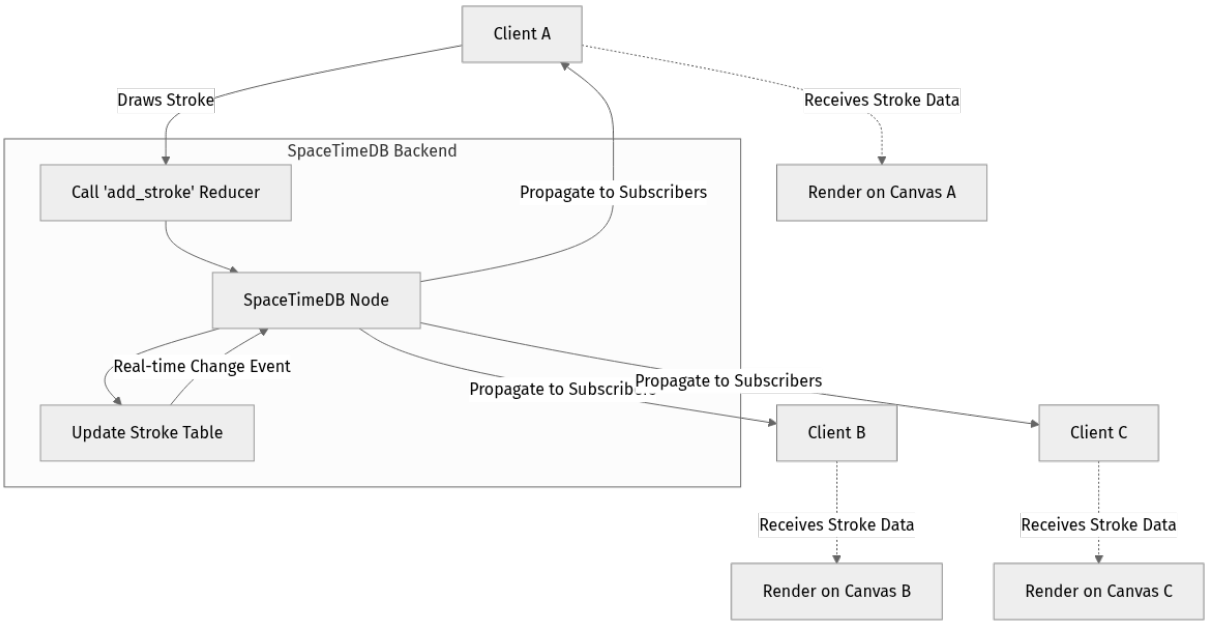


Figure 13.1: Real-time Stroke Synchronization with SpaceTimeDB

In this diagram, when Client A draws, the `add_stroke` reducer is invoked. SpaceTimeDB processes this, updates its internal database, and then

automatically pushes the new stroke data to all connected clients (A, B, and C), allowing them to render it on their respective canvases in real time.

4. Reducer Logic for Drawing Operations

Our primary reducer will be `add_stroke`. This reducer will accept the necessary details of a stroke (user ID, color, thickness, points array) and insert them into our `Stroke` table. Since SpaceTimeDB ensures deterministic execution and atomic updates, we don't have to worry about race conditions when multiple users try to draw simultaneously. Each stroke will be processed in order.

5. Client-Side Interaction (High-Level)

While this guide focuses on SpaceTimeDB, it's helpful to understand the client-side role. A typical frontend (e.g., a web application using HTML Canvas, React, or Vue) would: 1. Connect to the SpaceTimeDB instance. 2. Subscribe to changes in the `Stroke` table. 3. Implement drawing logic: * Detect mouse/touch events (down, move, up). * Collect points as the user draws. * On mouse/touch `up`, package the collected points, color, and thickness into a `Stroke` object. * Call the `add_stroke` reducer with this `Stroke` object. 4. When new stroke data arrives from SpaceTimeDB (either from the local client or another client), render it onto the HTML Canvas.

Step-by-Step Implementation

Let's start building our SpaceTimeDB backend for the collaborative whiteboard.

Step 1: Initialize Your SpaceTimeDB Project

First, ensure you have the SpaceTimeDB CLI installed (we'll assume `v2.1.0` as of 2026-03-14). If not, refer back to Chapter 2.

Open your terminal and create a new project:

```
# Create a new directory for our project
mkdir collaborative-whiteboard
cd collaborative-whiteboard

# Initialize a new SpaceTimeDB project
# We'll use the 'typescript' template for type safety
spacetimedb new --template typescript whiteboard_backend
```

This command creates a new directory `whiteboard_backend` inside `collaborative-whiteboard` with the basic SpaceTimeDB project structure.

Now, navigate into the newly created backend directory:

```
cd whiteboard_backend
```

Step 2: Define the Schema for Strokes and Presence

We need to define our `Stroke` and `UserPresence` tables. Open the `schema.st.js` file located in your `whiteboard_backend` directory.

Initially, it might contain some example schema. Let's replace it with our whiteboard-specific schema.

```
// whiteboard_backend/schema.st.js

import { Table, Reducer } from '@clockworklabs/spacetimedb-sdk';

/**
 * Represents a single drawing stroke on the whiteboard.
 */
@Table({
  table_name: 'Stroke',
})
export class Stroke {
  // A unique identifier for each stroke.
  // SpaceTimeDB automatically generates IDs for primary keys if not provided.
  @Table.PrimaryKey()
  id: string;

  // The ID of the user who created this stroke.
  user_id: string;

  // The color of the stroke (e.g., "#FF0000" for red).
  color: string;

  // The thickness of the stroke in pixels.
  thickness: number;

  // An array of points that make up the stroke.
  // Each point can be an object {x: number, y: number}.
  // We'll store this as a JSON string for simplicity within the schema,
  // but in TypeScript reducers, we'll parse/stringify it.
  points_json: string; // Storing as JSON string
}

/**
 * Tracks the presence of users on the whiteboard.
 */
@Table({
  table_name: 'UserPresence',
})
export class UserPresence {
  // The unique ID of the user. This is also the primary key.
  @Table.PrimaryKey()
  user_id: string;

  // The ID of the current whiteboard the user is on (useful for multiple
boards).
  current_board_id: string;

  // Timestamp of the last activity, useful for showing "online" status.
  last_active: number;
}

// Reducer declarations will go here later.
// For now, let's just define the tables.
```

Explanation: * We import `Table` and `Reducer` from the SpaceTimeDB SDK. * `@Table({ table_name: 'Stroke' })`: This decorator declares a new table named `Stroke`. * `@Table.PrimaryKey()`: Marks the `id` field as the primary key. SpaceTimeDB will ensure its uniqueness and can auto-generate values if not

explicitly set during insertion. * `user_id`, `color`, `thickness`: These are straightforward fields to store stroke properties. * `points_json: string`: This is a crucial design choice. While SpaceTimeDB's schema definition might not directly support complex nested types like `Array<{x: number, y: number}>` as a native column type, you can store such data as a JSON string. Your reducers and client-side code will be responsible for serializing (converting object to JSON string) before writing to the DB and deserializing (converting JSON string back to object) after reading from the DB. This is a common and flexible pattern for complex data. * The `UserPresence` table is self-explanatory, tracking `user_id`, `current_board_id`, and `last_active`.

After defining your schema, compile it using the SpaceTimeDB CLI:

```
spacetimedb compile
```

This command processes your `schema.st.js` and generates necessary TypeScript types and client-side SDK code in the `src/spacetimedb` directory. This generated code will be crucial for interacting with your database from reducers and clients.

Step 3: Implement the `add_stroke` Reducer

Now, let's create the reducer that will handle adding new strokes to our `Stroke` table.

Open the `src/modules/mod.ts` file. This is where your SpaceTimeDB reducers live.

Replace its content with the following:

```
// whiteboard_backend/src/modules/mod.ts

import { Reducer, SpacetimeDB, Identity } from '@clockworklabs/spacetimedb-
sdk';
import { Stroke } from '../spacetimedb/schema'; // Import our generated Stroke
type

/**
 * Adds a new drawing stroke to the whiteboard.
 * This reducer is called by clients when a user finishes drawing a stroke.
 *
 * @param identity The identity of the user calling this reducer.
 * @param user_id The ID of the user who created the stroke.
 * @param color The color of the stroke.
 * @param thickness The thickness of the stroke.
 * @param points_json A JSON string representing an array of points for the
stroke.
 */
@Reducer('add_stroke')
export function add_stroke(
  identity: Identity,
  user_id: string, // In a real app, you'd likely derive this from `identity`
  color: string,
  thickness: number,
  points_json: string
) {
  // Basic validation: Ensure points_json is not empty or malformed
  if (!points_json || points_json.length < 2) {
    SpacetimeDB.log.warn("Attempted to add an empty or invalid stroke.");
    return;
  }

  // Generate a unique ID for the new stroke.
  // We can use a combination of user_id and current timestamp, or a UUID
library.
  // For simplicity, let's use a basic timestamp-based ID here.
  const stroke_id = `${user_id}-${Date.now()}-${Math.random().toString(36).subs
tring(2, 9)}`;

  // Create a new Stroke object using the generated type
  const newStroke: Stroke = {
    id: stroke_id,
    user_id: user_id,
    color: color,
    thickness: thickness,
    points_json: points_json,
  };

  // Insert the new stroke into the Stroke table
  SpacetimeDB.insert(Stroke, newStroke);

  SpacetimeDB.log.info(`New stroke added by user ${user_id} with ID: ${stroke_i
d}`);
}

/**
 * Updates a user's presence on the whiteboard.
 * This reducer might be called periodically by clients to show active users.
 *
 * @param identity The identity of the user.
 * @param current_board_id The ID of the board the user is currently on.
```

```
 */
@Reducer('update_user_presence')
export function update_user_presence(
  identity: Identity,
  current_board_id: string
) {
  const user_id = identity.to_string(); // Use the identity as the user ID

  // Check if the user already exists in UserPresence
  const existingPresence = SpacetimeDB.filter(UserPresence, { user_id }).get_on
e();

  const now = Date.now();

  if (existingPresence) {
    // Update existing presence
    SpacetimeDB.update(UserPresence, { user_id }, { last_active: now, current_b
oard_id });
    SpacetimeDB.log.info(`User ${user_id} presence updated on board ${current_b
oard_id}.`);
  } else {
    // Insert new presence
    SpacetimeDB.insert(UserPresence, {
      user_id: user_id,
      current_board_id: current_board_id,
      last_active: now,
    });
    SpacetimeDB.log.info(`User ${user_id} joined board ${current_board_id}.`);
  }
}
```

Explanation: * `import { Stroke } from '../spacetimedb/schema';`: We import the `Stroke` class, which is a TypeScript type generated by `spacetimedb compile` based on our `schema.st.js`. This provides strong typing for our reducer logic! * `@Reducer('add_stroke')`: This decorator registers the `add_stroke` function as a SpaceTimeDB reducer, making it callable from clients. * `identity: Identity`: Every reducer receives the `identity` of the calling client. This is crucial for security and attribution. In a production app, you'd use `identity.to_string()` as the `user_id` to prevent clients from impersonating others. For simplicity here, we allow the client to pass `user_id`, but be aware of this security implication. * `stroke_id`: We generate a unique ID for the stroke. This is important for SpaceTimeDB's primary key and for client-side rendering (e.g., if a client needs to update an existing stroke, though we're only adding new ones here). * `SpacetimeDB.insert(Stroke, newStroke);`: This is the core action. It inserts our `newStroke` object into the `Stroke` table. SpaceTimeDB takes care of the rest, including real-time synchronization. * `@Reducer('update_user_presence')`: A second reducer for managing user presence. It uses `SpacetimeDB.filter` to check for existing presence and either `SpacetimeDB.update` or `SpacetimeDB.insert` accordingly. This demonstrates how to handle upsert-like logic.

After modifying your reducers, you need to compile them to generate the necessary bindings for the SpaceTimeDB module:

```
spacetimedb compile
```

Step 4: Run Your SpaceTimeDB Backend

Now that your schema and reducers are defined, you can start your SpaceTimeDB backend:

```
spacetimedb dev
```

You should see output indicating that SpaceTimeDB is running, typically on `ws://localhost:9000`. Keep this terminal window open. This command also deploys your compiled schema and reducers to the local SpaceTimeDB instance.

Step 5: Client-Side Interaction (Conceptual)

While a full frontend implementation is beyond the scope of this SpaceTimeDB guide, let's look at the critical SpaceTimeDB interaction points that your client-side JavaScript/TypeScript application would need.

You would install the SpaceTimeDB client SDK in your frontend project (e.g., using npm):

```
npm install @clockworklabs/spacetimedb-sdk
```

Then, your client-side code would look something like this (simplified for clarity):

```
// frontend/src/SpacetimeDBClient.ts (Conceptual file)

import { SpacetimeDBClient, Identity } from '@clockworklabs/spacetimedb-sdk';
import { add_stroke, update_user_presence, Stroke } from './spacetimedb/
client'; // Generated client SDK

const SPACETIMEDB_URI = 'ws://localhost:9000'; // Or your deployed SpaceTimeDB
instance
const BOARD_ID = 'main-whiteboard'; // A fixed ID for our simple whiteboard

let client: SpacetimeDBClient;
let currentIdentity: Identity;
let currentUserID: string; // This would typically come from an auth system

export async function connectToSpacetimeDB() {
  client = new SpacetimeDBClient(SPACETIMEDB_URI);

  client.onConnect(() => {
    console.log('Connected to SpaceTimeDB!');
    currentIdentity = client.identity;
    currentUserID = currentIdentity.to_string(); // Use the SpaceTimeDB
identity as the user ID

    // Subscribe to the Stroke table to receive real-time updates
    client.subscribe([
      { tableName: 'Stroke' },
      { tableName: 'UserPresence' }
    ]);

    // Send initial presence or update periodically
    update_user_presence(BOARD_ID);
  });

  client.onDisconnect(() => {
    console.log('Disconnected from SpaceTimeDB.');
  });

  // Listen for changes in the Stroke table
  client.on('Stroke', (strokes: Stroke[]) => {
    console.log('Received updated strokes:', strokes);
    // This is where your frontend would re-render the canvas
    // For example: `renderStrokesOnCanvas(strokes);`
  });

  // Listen for changes in UserPresence
  client.on('UserPresence', (presenceRecords) => {
    console.log('Updated user presence:', presenceRecords);
    // Update your UI to show who is online
  });

  await client.connect();
}

/**
 * Function to be called by the frontend drawing logic when a stroke is
complete.
 */
export function sendStrokeToSpacetimeDB(color: string, thickness: number, point
s: { x: number, y: number }[]) {
  if (!client || !currentUserID) {
    console.error("SpaceTimeDB client not connected or user ID not set.");
```

```
    return;
  }

  const points_json = JSON.stringify(points);

  // Call the 'add_stroke' reducer
  add_stroke(currentUserID, color, thickness, points_json);
  console.log('Called add_stroke reducer.');
}

// Example of how you might update presence periodically
setInterval(() => {
  if (client && currentUserID) {
    update_user_presence(BOARD_ID);
  }
}, 30000); // Every 30 seconds
```

Key client-side takeaways: * `SpacetimeDBClient`: The main entry point for connecting. * `client.onConnect()`: Establish connection and get `identity`. * `client.subscribe()`: Crucially, subscribe to the `Stroke` and `UserPresence` tables to receive real-time updates. * `client.on('Stroke', (strokes: Stroke[]) => { ... });`: This event listener fires whenever the `Stroke` table changes. The `strokes` array will contain the entire current state of the `Stroke` table, allowing your frontend to re-render. * `add_stroke(currentUserID, color, thickness, points_json);`: This directly calls your server-side reducer. The generated client SDK (`./spacetimedb/client`) provides these functions.

With this setup, your frontend would handle the drawing on an HTML Canvas, collect the points, and then simply call `sendStrokeToSpacetimeDB`. SpaceTimeDB would then handle the persistence and real-time distribution to all other connected clients, making your whiteboard collaborative!

Mini-Challenge: Clear the Whiteboard

You've successfully built the core logic for adding strokes. Now, let's add a feature to manage the whiteboard's state: clearing all strokes.

Challenge: Create a new SpaceTimeDB reducer called `clear_whiteboard`. This reducer should delete all existing strokes from the `Stroke` table.

Hint: SpaceTimeDB's `SpacetimeDB.delete()` function can accept a filter to delete multiple rows. If you want to delete all rows from a table, what would your filter look like? (Think about an empty filter or a filter that matches everything).

What to observe/learn: How to perform bulk delete operations using SpaceTimeDB reducers, which is essential for managing dynamic data sets.

Click for Solution (after you've tried it!)

```
// whiteboard_backend/src/modules/mod.ts (Add this to your existing file)

// ... existing imports and reducers ...

/**
 * Clears all drawing strokes from the whiteboard.
 * This reducer is typically called by an authorized user (e.g., moderator).
 *
 * @param identity The identity of the user calling this reducer.
 */
@Reducer('clear_whiteboard')
export function clear_whiteboard(identity: Identity) {
  // In a real application, you'd add authorization logic here:
  // if (!isModerator(identity)) {
  //   SpacetimeDB.log.warn(`Unauthorized attempt to clear whiteboard by ${identity.to_string()}`);
  //   return;
  // }

  // To delete all rows from a table, you can pass an empty filter object `{}`
  // or use a filter that always evaluates to true.
  // The simplest way to delete all is to provide an empty filter.
  SpacetimeDB.delete(Stroke, {}); // Deletes all rows from the Stroke table

  SpacetimeDB.log.info(`Whiteboard cleared by ${identity.to_string()}.`);
}
```

Explanation of Solution: By calling `SpacetimeDB.delete(Stroke, {});`, we tell SpaceTimeDB to delete all entries from the `Stroke` table that match an empty filter. An empty filter matches all entries, effectively clearing the table. Remember to re-run `spacetimedb compile` and `spacetimedb dev` after adding this reducer! On the client side, you would simply call `clear_whiteboard()` from the generated client SDK.

Common Pitfalls & Troubleshooting

1. **Complex `points_json` Handling:**

- **Pitfall:** Forgetting to `JSON.stringify()` the `points` array before sending it to the reducer (from the client) or before inserting it into the `Stroke` table (within the reducer). Similarly, forgetting to `JSON.parse()` the `points_json` string after receiving it on the client.
- **Troubleshooting:** Check your client-side `sendStrokeToSpacetimeDB` function and your `add_stroke` reducer. Use `console.log()` to inspect the `points` data just before it's sent/inserted and immediately after it's

received/read. Ensure it's a valid JSON string when interacting with the database.

1. **Reducer Idempotency for Real-time Updates:**

- **Pitfall:** While `add_stroke` is inherently additive, if you were to design an `update_stroke` reducer, you'd need to ensure it can be safely called multiple times without unintended side effects. For example, if a client attempts to update a stroke that has already been deleted.
- **Troubleshooting:** Always retrieve the current state of the row you intend to update or delete within your reducer. Perform checks like `if (existingStroke)` before attempting an `update` or `delete`. This ensures your reducer operates on the actual current state.

1. **Client-Side Rendering Performance with Many Strokes:**

- **Pitfall:** As the number of strokes grows, re-rendering the entire canvas every time a new stroke arrives can become slow, especially on less powerful devices.
- **Troubleshooting:** This is primarily a frontend optimization, but SpaceTimeDB's data model allows for it. Instead of clearing and redrawing everything, your client-side `client.on('Stroke', ...)` listener could be smarter: * Maintain a local cache of strokes. * When SpaceTimeDB sends updates, identify only the new or changed strokes. * Only draw the new/changed strokes, or use a double-buffering technique on the canvas. * Periodically "bake" older strokes into a static background layer to reduce the number of active elements to render.

Summary

Phew! You've just completed a significant project milestone: laying the foundation for a real-time collaborative whiteboard using SpaceTimeDB.

Here are the key takeaways from this chapter:

- **Schema Design for Dynamic Data:** You learned how to model complex, dynamic data like drawing strokes, including using JSON strings for array-of-objects fields.
- **Event-Driven Reducers:** You implemented `add_stroke` and `update_user_presence` reducers to handle core drawing and presence logic, demonstrating how SpaceTimeDB processes state changes.

- **Real-time Synchronization in Practice:** You saw how SpaceTimeDB automatically synchronizes database changes to all connected clients, making collaborative features incredibly efficient to build.
- **Client-Side Interaction:** You gained a conceptual understanding of how a frontend application connects, subscribes, and calls reducers to achieve real-time collaboration.
- **Bulk Operations:** The mini-challenge introduced you to performing bulk deletes with reducers, a crucial skill for managing data.

This project highlights SpaceTimeDB's strength in creating highly interactive and collaborative applications where shared, real-time state is paramount. You've now got a solid foundation for building your own multiplayer games, collaborative editors, or real-time dashboards!

In the next chapter, we'll dive into more advanced topics like security models and authentication integration, which are critical for production-ready collaborative applications.

References

1. SpaceTimeDB Official Documentation: https://spacetimedb.com/docs
2. SpaceTimeDB GitHub Repository: https://github.com/clockworklabs/SpacetimeDB
3. SpaceTimeDB CLI Releases: https://github.com/clockworklabs/SpacetimeDB/releases
4. MDN Web Docs - JSON.stringify(): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
5. MDN Web Docs - HTML Canvas API: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Chapter 14: Project: Developing a Simple Multiplayer Game

Introduction: Bringing Games to Life with SpaceTimeDB

Welcome to Chapter 14! In this exciting chapter, we're going to put all our SpaceTimeDB knowledge to the test by building a simple, yet engaging, real-time multiplayer game. Imagine a canvas where multiple players can move their unique cursors or avatars around, and everyone sees everyone else's movements instantly. That's the magic we're aiming for!

This project is more than just a game; it's a practical demonstration of how SpaceTimeDB's core strengths—its unified database and backend logic, real-time synchronization, and deterministic reducers—make it an ideal platform for collaborative and interactive applications. By the end of this chapter, you'll have a clear understanding of how to manage shared game state, process player actions, and update the game world in real-time across all connected clients.

Before we dive in, make sure you're comfortable with the SpaceTimeDB CLI, defining schemas, writing reducers, and connecting a basic JavaScript client from previous chapters. We'll be building on those foundations to create our multiplayer masterpiece!

Core Concepts for Multiplayer Games

Building a multiplayer game with SpaceTimeDB feels remarkably intuitive because its architecture naturally aligns with game development needs. Let's break down the key concepts we'll leverage.

Game State Management with SpaceTimeDB

In a multiplayer game, the "game state" is everything that describes the current situation of the game: player positions, scores, item locations, and so on. The challenge is keeping this state consistent and synchronized across all players and the server.

SpaceTimeDB excels here. It acts as the single source of truth for your entire game state. When a player makes a move, it's not just updating their local

screen; it's submitting a "transaction" (a reducer call) to SpaceTimeDB. SpaceTimeDB processes this, updates its internal state, and then automatically pushes those changes out to all subscribed clients. This real-time, consistent shared state is the bedrock of our multiplayer experience.

Player Representation: The `Player` Table

Every player in our game needs to be represented in our database. We'll create a `Player` table to store essential information for each active participant. What kind of information might a player need? At a minimum, an ID, a name, and their position on the game board. We might also want a color to distinguish them visually.

Game Actions as Reducers

How do players interact with the game? By performing actions! Moving their avatar, clicking a button, picking up an item—these are all actions. In SpaceTimeDB, these actions map perfectly to **reducers**.

A reducer takes the current game state and an action's parameters, then produces a new game state. Because reducers are deterministic and executed on the SpaceTimeDB server, they ensure that every client's view of the game state remains consistent. If two players try to move at the same time, SpaceTimeDB handles the order of operations, and all clients see the final, consistent result.

Real-time UI Updates

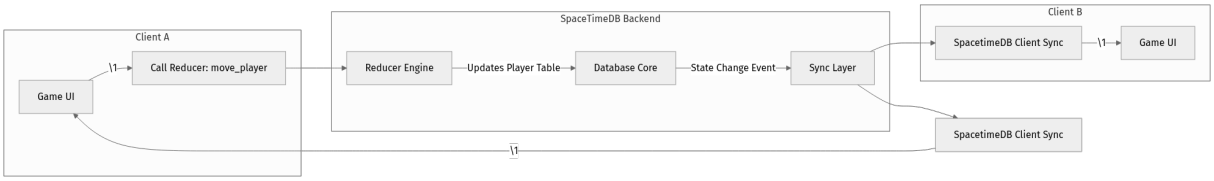
Once SpaceTimeDB updates its state based on a reducer, how do clients know to update their screens? Through **subscriptions**! Our frontend client will subscribe to the `Player` table. Any time a player's position changes (because another player moved them via a reducer), SpaceTimeDB pushes that update to our client. Our client then receives this data and re-renders the game world to reflect the latest state. This push-based, event-driven synchronization is key to a smooth real-time experience.

Client-Side Framework Choice

For this simple example, we'll use plain HTML, CSS, and JavaScript. This keeps the focus squarely on SpaceTimeDB's integration without introducing the complexities of a frontend framework like React or Vue. However, the principles we learn here apply directly to any frontend framework you might choose for a more complex game.

Architectural Flow

Let's visualize the flow of data and actions in our multiplayer game:



- **Client A** takes user input (e.g., mouse movement).
- It calls a SpaceTimeDB reducer (e.g., `move_player`) with new coordinates.
- The **SpaceTimeDB Backend** receives the reducer call, executes the `move_player` logic, and updates the `Player` table in its database core.
- Upon a state change, SpaceTimeDB's synchronization layer automatically pushes these updates to **all connected clients**, including Client A and Client B.
- **Clients A and B** receive the updates and re-render their game UI to reflect the new player positions.

Step-by-Step Implementation: Building Our Multiplayer Cursor Game

Let's get our hands dirty and build this!

Step 1: Initialize Your SpaceTimeDB Project

First, ensure you have the SpaceTimeDB CLI installed. As of 2026-03-14, we're using SpaceTimeDB CLI `v2.x`.

Open your terminal and create a new project:

```
stdb new multiplayer-game
cd multiplayer-game
```

This command creates a new directory `multiplayer-game` with the basic SpaceTimeDB project structure.

Step 2: Define the Game Schema

Now, let's define our `Player` table. Open `stdb/schema.stdb` and add the following:

```
// stdb/schema.stdb

// Define the Player table to store information about each connected player.
// `#[primarykey]` ensures each player has a unique ID.
// `#[autoinc]` automatically assigns a new ID for new players.
#[primarykey(id)]
#[autoinc(id)]
table Player {
    id: u64, // Unique identifier for the player
    name: String, // Player's display name
    x: f32, // X-coordinate on the game board
    y: f32, // Y-coordinate on the game board
    color: String, // A color string (e.g., "#RRGGBB") for their avatar
}
```

Explanation: * `#[primarykey(id)]`: Designates `id` as the primary key, ensuring uniqueness and efficient lookups. * `#[autoinc(id)]`: SpaceTimeDB will automatically assign a new, incrementing `u64` value to `id` when a new `Player` row is inserted without specifying an `id`. This is incredibly useful for new players joining. * `table Player { ... }`: Defines our `Player` table with fields for `id`, `name`, `x` (position), `y` (position), and `color`. We use `f32` for coordinates to allow for fractional positions, which can make movement smoother.

Step 3: Implement Reducers for Player Actions

Next, we'll create a module for our player-related reducers. Create a new file `stdb/modules/player_actions.stdb`:

```
// stdb/modules/player_actions.stdb

// Import the Player table definition from our schema.
use crate::Player;

// Define a reducer to create a new player when they join the game.
#[reducer]
fn create_player(ctx: ReducerContext, name: String, initial_x: f32, initial_y:
f32, color: String) -> Player {
    // Insert a new row into the Player table.
    // The 'id' field will be automatically assigned due to #[autoinc] in the
schema.
    Player::insert(Player {
        id: 0, // Placeholder, will be ignored due to #[autoinc]
        name,
        x: initial_x,
        y: initial_y,
        color,
    })
}

// Define a reducer to update a player's position.
#[reducer]
fn move_player(ctx: ReducerContext, player_id: u64, new_x: f32, new_y: f32) {
    // Find the player by their ID.
    // `Player::filter_by_id()` returns an iterator. `.next()` gets the first
match.
    // `.expect()` will panic if the player is not found, which is okay for
this simple example.
    let mut player = Player::filter_by_id(player_id).next().expect("Player not
found.");

    // Update the player's x and y coordinates.
    player.x = new_x;
    player.y = new_y;

    // Save the updated player back to the table.
    player.update();
}
```

Explanation: * `use crate::Player;`: Imports our `Player` table definition so we can interact with it. * `#[reducer] fn create_player(...)`: This reducer handles adding new players. * It takes `ReducerContext` (standard for all reducers), `name`, `initial_x`, `initial_y`, and `color`. * `Player::insert(...)`: Creates a new `Player` entry. We pass `id: 0` as a placeholder because `#[autoinc]` will provide the actual ID. * It returns the newly created `Player` object. * `#[reducer] fn move_player(...)`: This reducer updates a player's position. * It takes the `player_id` and the `new_x`, `new_y` coordinates. * `Player::filter_by_id(player_id).next().expect(...)`: This is how we retrieve a specific player by their primary key. * `player.x = new_x; player.y = new_y;`: We modify the retrieved `player` object. * `player.update();`: This persists the changes back to the `Player` table. Without `update()`, the changes would not be saved.

Step 4: Run SpaceTimeDB and Generate Client Libraries

Now that our schema and reducers are defined, let's compile our SpaceTimeDB module and start the local server.

First, from your project root, generate the client libraries (including JavaScript for our frontend):

```
stdb generate
```

This command compiles your Rust modules and generates client-side code in the `target/` directory. You'll find a JavaScript client in `target/multiplayer-game.js`.

Next, run the SpaceTimeDB development server:

```
stdb run
```

Keep this terminal window open. Your SpaceTimeDB instance is now running and listening for client connections, typically on `ws://localhost:9000`.

Step 5: Build the Frontend (HTML & JavaScript)

Let's create a simple HTML page and a JavaScript file to connect to SpaceTimeDB and visualize our game.

Create `index.html` in the root of your `multiplayer-game` directory:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Multiplayer Cursor Game</title>
    <style>
        body { margin: 0; overflow: hidden; background-color: #222; font-
family: sans-serif; }
        #game-area {
            position: relative;
            width: 100vw;
            height: 100vh;
            cursor: none; /* Hide default cursor */
        }
        .player-cursor {
            position: absolute;
            width: 30px;
            height: 30px;
            border-radius: 50%;
            border: 2px solid white;
            box-sizing: border-box;
            transform: translate(-50%,
-50%); /* Center the cursor on its (x,y) */
            transition: transform 0.05s linear; /* Smooth movement */
            pointer-events: none; /* Allow mouse events to pass through */
            display: flex;
            align-items: center;
            justify-content: center;
            font-size: 10px;
            color: white;
            text-shadow: 0 0 3px black;
        }
    </style>
</head>
<body>
    <div id="game-area"></div>

    <!-- The generated SpacetimeDB client library -->
    <script type="module" src="./target/multiplayer-game.js"></script>
    <!-- Our game logic -->
    <script type="module" src="./game.js"></script>
</body>
</html>
```

Explanation: * We define a `#game-area` div that will serve as our canvas. * `.player-cursor` styles are for the visual representation of each player. * Crucially, we include `target/multiplayer-game.js` (our generated SpaceTimeDB client) and `game.js` (our custom game logic). Note the `type="module"` for both.

Now, create `game.js` in the root of your `multiplayer-game` directory:

```
// game.js

import { SpacetimeDBClient, multiplayer_game } from './target/multiplayer-
game.js';

const gameArea = document.getElementById('game-area');
const SPDB_SERVER_URL = 'ws://localhost:9000'; // SpaceTimeDB server address

let myPlayerId = null;
const activePlayers = new Map(); // Map to store player DOM elements by ID

// Function to generate a random hex color
function getRandomColor() {
    const letters = '0123456789ABCDEF';
    let color = '#';
    for (let i = 0; i < 6; i++) {
        color += letters[Math.floor(Math.random() * 16)];
    }
    return color;
}

// Connect to SpacetimeDB
async function connectToSpacetimeDB() {
    console.log(`Connecting to SpaceTimeDB at ${SPDB_SERVER_URL}...`);
    await SpacetimeDBClient.connect(SPDB_SERVER_URL);
    console.log('Connected to SpaceTimeDB!');

    // Generate a random name and color for this player
    const playerName = `Player_${Math.floor(Math.random() * 1000)}`;
    const playerColor = getRandomColor();

    // Call the create_player reducer to add ourselves to the game
    // Initial position is center of the screen
    multiplayer_game.create_player(playerName, window.innerWidth / 2, window.in
nerHeight / 2, playerColor)
        .then(newPlayer => {
            myPlayerId = newPlayer.id; // Store our player ID
            console.log(`My player ID is: ${myPlayerId}`);
            // We don't need to add our own cursor here; the subscription will
handle it.
        })
        .catch(error => {
            console.error('Failed to create player:', error);
        });

    // Subscribe to the Player table to get real-time updates for all players
    SpacetimeDBClient.subscribe([
        { tableName: 'Player' }
    ]);

    // Listen for changes to the Player table
    multiplayer_game.onPlayerUpdate((player, oldPlayer) => {
        if (player) {
            // Player created or updated
            let playerElement = activePlayers.get(player.id);
            if (!playerElement) {
                // New player, create their cursor element
                playerElement = document.createElement('div');
                playerElement.classList.add('player-cursor');
                playerElement.id = `player-${player.id}`;
                gameArea.appendChild(playerElement);
```

```
                activePlayers.set(player.id, playerElement);
            }
            // Update position and color
            playerElement.style.left = `${player.x}px`;
            playerElement.style.top = `${player.y}px`;
            playerElement.style.backgroundColor = player.color;
            playerElement.textContent = player.name;
        } else if (oldPlayer) {
            // Player deleted (e.g., disconnected or removed)
            const playerElement = activePlayers.get(oldPlayer.id);
            if (playerElement) {
                playerElement.remove();
                activePlayers.delete(oldPlayer.id);
            }
        }
    });

    // Handle mouse movement to update our player's position
    gameArea.addEventListener('mousemove', (event) => {
        if (myPlayerId !== null) {
            // Call the move_player reducer
            multiplayer_game.move_player(myPlayerId, event.clientX, event.clien
tY);
        }
    });
}

connectToSpacetimeDB();

// Handle window resize to adjust initial position for new players
window.addEventListener('resize', () => {
    if (myPlayerId !== null) {
        // Optionally, update own position on resize or just let new players
spawn correctly
    }
});
```

Explanation: 1. **Import Client:** `import { SpacetimeDBClient, multiplayer_game } from './target/multiplayer-game.js';` imports the necessary client library. `multiplayer_game` is the object containing our generated reducer functions. 2. **`connectToSpacetimeDB()`:** * `SpacetimeDBClient.connect(SPDB_SERVER_URL)`: Establishes a WebSocket connection to your running SpaceTimeDB server. * `multiplayer_game.create_player(...)`: After connecting, we immediately call our `create_player` reducer. This adds our player to the `Player` table in SpaceTimeDB. We store the returned `id` as `myPlayerId`. * `SpacetimeDBClient.subscribe([{ tableName: 'Player' }])`: This is the magic! We tell SpaceTimeDB we want to receive real-time updates for any changes to the `Player` table. * `multiplayer_game.onPlayerUpdate((player, oldPlayer) => { ... })`: This event listener fires whenever a `Player` row is inserted, updated, or deleted. * If `player` is present, it's a new player or an update. We create a new `div` if it's a new player, or update the existing one's

position and color. * If `oldPlayer` is present and `player` is null, it means a player was deleted (e.g., disconnected). We remove their cursor element. * `gameArea.addEventListener('mousemove', ...)`: When our mouse moves, we call `multiplayer_game.move_player()` with our `myPlayerId` and the new mouse coordinates. This sends the update to SpaceTimeDB.

Step 6: Test Your Multiplayer Game

1. Ensure your `stdb run` terminal is still active.
2. Open `index.html` in your web browser. You can typically do this by dragging the file into your browser or using a simple local web server (like `python -m http.server` if you have Python installed).
3. You should see a colored circle representing your player.
4. Open `index.html` in a second browser tab or window.
5. Voila! You should now see two distinct cursors moving independently, each controlled by a different browser tab. Each tab will see the other player's movements in real-time.

Congratulations! You've just built a real-time multiplayer game with SpaceTimeDB!

Mini-Challenge: Adding a Score Counter

Let's make our game a bit more interactive. How about adding a score to each player and a way to increment it?

Challenge: 1. **Update the `Player` Schema:** Add a `score: u32` field to the `Player` table, initialized to `0`. 2. **Create a New Reducer:** Implement a reducer called `increment_score(ctx: ReducerContext, player_id: u64)` that takes a player's ID and increments their `score` by 1. 3. **Modify the Frontend:** * Display the `score` next to the player's name in their cursor element. * Add an event listener (e.g., a click event on the `gameArea` or a specific key press) that, when triggered, calls your new `increment_score` reducer for your own player.

Hint: * Remember to run `stdb generate` and restart `stdb run` after changing `stdb/schema.stdb` and `stdb/modules/player_actions.stdb`. * The `onPlayerUpdate` callback in `game.js` will automatically give you the updated `Player` object with the new `score` value.

What to Observe/Learn: * How seamlessly SpaceTimeDB handles schema evolution and new reducer logic. * The instant propagation of score updates to all connected clients, demonstrating shared state beyond just position.

Common Pitfalls & Troubleshooting

Even simple projects can hit snags. Here are a few common issues and how to resolve them:

1. **SpaceTimeDB Server Not Running or Not Accessible:**

- **Symptom:** Your browser console shows "WebSocket connection failed" or "Connection refused".
- **Fix:** Ensure you have `stdb run` actively running in a terminal. Check the port (default is 9000). If you changed it, update `SPDB_SERVER_URL` in `game.js`. Firewall issues could also block the connection.

1. **Schema or Reducer Changes Not Reflected:**

- **Symptom:** Reducer calls fail with "reducer not found" errors, or new fields are missing from `Player` objects.
- **Fix:** Did you run `stdb generate` after modifying `stdb/schema.stdb` or `stdb/modules/player_actions.stdb`? And did you restart `stdb run`? These steps are crucial to recompile and reload your SpaceTimeDB module.

1. **Client-Side Subscription Issues:**

- **Symptom:** Players can move, but you don't see other players' movements, or new players don't appear.
- **Fix:** Double-check `SpacetimeDBClient.subscribe([{ tableName: 'Player' }])` in `game.js`. Is the table name correct? Also, ensure `multiplayer_game.onPlayerUpdate` is correctly implemented and processing both `player` (new/updated) and `oldPlayer` (deleted) cases.

1. **Frontend Rendering Problems:**

- **Symptom:** Data is coming in, but nothing appears on screen or elements are mispositioned.
- **Fix:** Use your browser's developer tools. Inspect the HTML elements for `player-cursor` divs. Are they being created? Do their `left` and `top` styles have valid `px` values? Check for JavaScript errors in the console related to DOM manipulation.

1. **Reducer Logic Errors (Non-Deterministic):**

- **Symptom:** Very rare in this simple example, but in more complex games, if a reducer's outcome depends on external factors (like `Math.random()` or

current time without `ctx`), different SpaceTimeDB replicas could produce different states.

- **Fix:** Always ensure reducers are pure functions of their inputs and the current SpaceTimeDB state. If randomness is needed, generate it once and pass it as a reducer parameter or use SpaceTimeDB's `ReducerContext` for any time-related operations if available (though typically, time-sensitive game logic is handled by clients or a dedicated game server that then updates SpaceTimeDB).

Summary: Your First Multiplayer Game!

Phew! What an exciting journey. You've just built a fully functional, real-time multiplayer game powered by SpaceTimeDB. Let's recap the key takeaways:

- **Unified Backend:** SpaceTimeDB seamlessly combines your database and backend game logic into a single, cohesive system.
- **Deterministic Reducers:** Player actions are transformed into deterministic reducers, ensuring consistent state updates across all clients.
- **Real-time Synchronization:** Subscriptions to tables like `Player` allow SpaceTimeDB to automatically push state changes to all connected clients, enabling instant updates for everyone.
- **Simplified Game Development:** By abstracting away much of the complex networking and synchronization, SpaceTimeDB lets you focus on game logic.
- **Practical Application:** This project demonstrated how SpaceTimeDB is perfectly suited for interactive, collaborative, and real-time applications, from games to dashboards.

You now have a solid foundation for building more complex multiplayer experiences. In the next chapter, we'll delve into more advanced topics like concurrency handling and transactions, which are essential for robust game logic and preventing race conditions.

References

- SpaceTimeDB Official Documentation
- SpaceTimeDB GitHub Repository
- MDN Web Docs: WebSockets

- MDN Web Docs: EventTarget

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Chapter 15: Debugging, Testing, and Observability in SpaceTimeDB

Introduction

Welcome to Chapter 15! As we've journeyed through the capabilities of SpaceTimeDB, building real-time, collaborative applications, you might have encountered situations where things didn't quite work as expected. This is a natural part of software development, and it highlights the critical importance of debugging, testing, and observability.

In this chapter, we'll equip you with the essential skills and tools to confidently diagnose problems, ensure the correctness of your SpaceTimeDB logic, and monitor your applications in production. We'll explore strategies for both server-side (reducer) and client-side debugging, delve into writing robust unit and integration tests, and discuss how to establish comprehensive observability using logs, metrics, and tracing. By the end of this chapter, you'll not only be able to build powerful SpaceTimeDB applications but also maintain and scale them with confidence.

To get the most out of this chapter, you should have a solid understanding of SpaceTimeDB's core concepts, including schema definition, reducers, and client-side interaction, as covered in previous chapters. Let's dive in and make your SpaceTimeDB applications rock-solid!

Core Concepts

Building reliable real-time systems with SpaceTimeDB requires a structured approach to identifying and preventing issues. This section lays out the fundamental concepts for debugging, testing, and observing your applications.

Debugging SpaceTimeDB Applications

Debugging is the process of finding and fixing errors or bugs in your code. With SpaceTimeDB, debugging involves both your server-side Rust reducers and your client-side application logic.

Server-Side Reducer Debugging

SpaceTimeDB's reducers are written in Rust, which offers excellent debugging capabilities. When a reducer executes, it runs within the SpaceTimeDB server environment.

1. **`spacetime-cli logs`**: The most straightforward way to see what's happening inside your reducers is through the `spacetime-cli`'s logging functionality. When you run `spacetime-cli dev` or `spacetime-cli deploy`, the server's output, including any `println!` or `dbg!` macros from your Rust code, will be streamed to your terminal.
    - `println!`: This macro prints formatted text to standard output, similar to `console.log()` in JavaScript. It's useful for inspecting variable values at specific points in your code.
    - `dbg!`: This macro is a powerful debugging tool in Rust. It prints the file name, line number, and expression of its argument, along with its value and a pretty-printed representation. It's especially handy for quickly inspecting complex data structures.
2. **Rust `log` Crate**: For more structured and production-ready logging, you can integrate the `log` crate into your reducer modules. This allows you to log messages with different levels (info, debug, warn, error) and can be configured to integrate with external logging systems in a deployed environment.
3. **Rust Debugger Integration**: For deep introspection, you can attach a debugger (like `rust-lldb` or `rust-gdb`, often integrated into IDEs like VS Code via extensions like Rust Analyzer) to your SpaceTimeDB server process. This allows you to set breakpoints, step through code line by line, and inspect the call stack and variable states in real-time. This is typically done against a locally running SpaceTimeDB instance.

Client-Side Debugging

Your client application interacts with SpaceTimeDB through its client libraries (e.g., TypeScript/JavaScript). Standard frontend debugging techniques apply here.

1. **Browser Developer Tools**: For web clients, the browser's developer tools (Console, Network, Sources tabs) are your best friends. You can set breakpoints in your JavaScript/TypeScript code, inspect network requests (including WebSocket messages to/from SpaceTimeDB), and observe the client-side state.
2. **Client Library Logging**: SpaceTimeDB client libraries often have configurable logging levels. Enabling verbose logging can provide insights

into connection status, subscription updates, and any errors received from the server.

Testing SpaceTimeDB Logic

Testing is about verifying that your code behaves as expected under various conditions. For SpaceTimeDB, this means ensuring your reducers are deterministic and correct, and that your client-server interactions are robust.

Why Testing is Crucial for SpaceTimeDB

SpaceTimeDB's core principle of deterministic reducers means that for a given initial state and a sequence of reducer calls, the final state will always be the same. This determinism makes testing highly effective and crucial:

- **Reliability**: Ensures your shared state remains consistent across all clients.
- **Prevent Regressions**: Catches unintended side effects when modifying existing code.
- **Confidence**: Allows you to refactor and evolve your schema and reducers without fear.

Types of Testing

1. **Unit Testing Reducers**: This focuses on testing individual reducer functions in isolation. You provide a mock initial state, call the reducer with specific arguments, and assert that the resulting state change (or error) is correct. Rust's built-in testing framework (`cargo test`) is perfect for this.
2. **Integration Testing**: This involves testing the interaction between different components. For SpaceTimeDB, this could mean:
    - Testing a client connecting to a local SpaceTimeDB instance.
    - Calling reducers from the client and observing the resulting state changes propagated back to the client.
    - Verifying schema evolution and data migrations.
3. **End-to-End (E2E) Testing**: Simulating real user interactions across your entire application stack, from the UI to SpaceTimeDB and back. This often involves tools like Playwright or Cypress for web applications, interacting with your SpaceTimeDB client.

Observability in Production

Once your SpaceTimeDB application is deployed, debugging with `println!` isn't enough. Observability is the ability to understand the internal state of a system by examining its external outputs. It's crucial for diagnosing issues in production

environments without needing to deploy new code. The three pillars of observability are logs, metrics, and traces.

1. Logging

Logs are discrete, time-stamped records of events that happen within your application.

- **Structured Logging**: Instead of plain text, log data as structured JSON. This makes it machine-readable and easier to query and analyze in centralized logging systems (e.g., ELK Stack, Splunk, DataDog).
- **Contextual Information**: Include relevant context in your logs, such as `user_id`, `reducer_name`, `table_name`, `error_code`, or `request_id`.
- **SpaceTimeDB Specifics**: Your reducers should emit logs for important events (e.g., data mutations, authorization failures, complex logic branches). Your client-side code should log connection events, subscription changes, and client-side errors.

2. Metrics

Metrics are numerical measurements collected over time, often aggregated. They provide a high-level view of your system's health and performance.

- **Key SpaceTimeDB Metrics**:
- **Reducer Execution Time**: Average, p95, p99 latency for each reducer.
- **Reducer Call Count**: How often each reducer is invoked.
- **Client Connection Count**: Number of active WebSocket connections.
- **Data Change Rate**: How many rows are inserted/updated/deleted per second.
- **Subscription Latency**: Time taken for state changes to propagate to clients.
- **Error Rates**: Number of reducer errors or client-side errors.
- **Monitoring Tools**: Metrics are typically collected by agents and sent to monitoring dashboards (e.g., Prometheus/Grafana, DataDog, New Relic) for visualization and alerting.

3. Tracing (Distributed Tracing)

Traces represent the end-to-end journey of a request or operation through a distributed system. While SpaceTimeDB itself provides a unified backend, your

overall application might involve other services (e.g., an authentication service, external APIs).

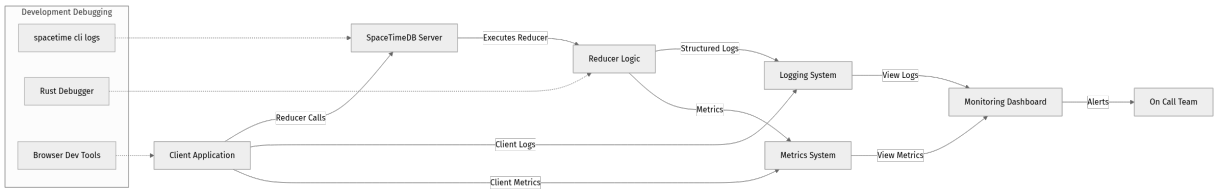
- **Correlation IDs**: Propagate a unique correlation ID from the initial client request through your SpaceTimeDB reducer calls and any subsequent external service calls. This allows you to link related log entries and metric events.
- **OpenTelemetry**: Consider using a standard like OpenTelemetry to instrument your client and reducer code (if SpaceTimeDB integrates with it directly or allows custom instrumentation) to generate and export traces.

Alerting

Alerting is the act of notifying a team when a metric crosses a predefined threshold or an important log event occurs.

- **Critical Alerts**: For production-impacting issues (e.g., high error rates, server down).
- **Warning Alerts**: For potential issues that need investigation (e.g., increasing latency, resource utilization spikes).

Observability Architecture



- **Client Application**: Your frontend, game client, or other application using the SpaceTimeDB client library.
- **SpaceTimeDB Server**: The core SpaceTimeDB instance hosting your schema and reducers.
- **Reducer Logic**: The Rust code you write that modifies the database state.
- **Logging System**: Collects structured logs from both client and server.
- **Metrics System**: Collects numerical performance and health metrics.
- **Monitoring Dashboard**: Visualizes logs and metrics, allowing for querying and analysis.
- **On-Call Team**: Receives alerts when critical issues arise.
- **Debugging Tools**: Used during development for interactive problem-solving.

This diagram illustrates how development debugging tools integrate with the system during development, and how logs and metrics flow into dedicated systems for production observability.

Step-by-Step Implementation

Let's get practical and implement some debugging and testing techniques.

Setting up a Basic Project (if not already done)

For this chapter, we'll assume you have a basic SpaceTimeDB project set up. If not, quickly create one:

```
# Ensure you have spacetime-cli installed (latest stable as of 2026-03-14, e.g., v2.1.0)
# Check official docs for installation: https://spacetimedb.com/docs/getting-started/installation
# For example, `curl -sL https://install.spacetimedb.com | bash`

spacetime-cli new my_debug_test_app
cd my_debug_test_app
```

Now, let's define a simple schema and reducer. Open `src/lib.rs` and replace its content:

```rust
// src/lib.rs
use spacetimedb::{spacetimedb, table, ReducerContext};

// Define a simple Counter table
#[spacetimedb(table)]
pub struct Counter {
    #[primarykey]
    pub id: u384, // Using u384 for a unique identifier, common in SpaceTimeDB
for primary keys
    pub value: u32,
}

// Define a reducer to increment the counter
#[spacetimedb(reducer)]
pub fn increment_counter(ctx: ReducerContext, id: u384, amount: u32) ->
Result<(), String> {
    // Attempt to get the counter by ID
    let mut counter = Counter::filter_by_id(&id)
        .ok_or_else(|| format!("Counter with id {} not found.", id))?;

    // Increment the value
    counter.value = counter.value.checked_add(amount)
        .ok_or_else(|| "Counter overflowed!".to_string())?;

    // Update the counter in the database
    counter.update();

    Ok(())
}

// Define a reducer to create a new counter
#[spacetimedb(reducer)]
pub fn create_counter(ctx: ReducerContext, id: u384, initial_value: u32) -> Res
ult<(), String> {
    if Counter::filter_by_id(&id).is_some() {
        return Err(format!("Counter with id {} already exists.", id));
    }
    Counter::insert(Counter { id, value: initial_value });
    Ok(())
}
```

Build and run your SpaceTimeDB instance:

```
spacetime-cli dev
```

You should see output indicating the server is running and your module is deployed.

Step 1: Server-Side Debugging with `println!` and `dbg!`

Let's add some debugging statements to our `increment_counter` reducer.

Open `src/lib.rs` again and modify the `increment_counter` reducer:

```
// src/lib.rs (modifications within increment_counter reducer)
// ... existing code ...

#[spacetimedb(reducer)]
pub fn increment_counter(ctx: ReducerContext, id: u384, amount: u32) ->
Result<(), String> {
    // Debugging step 1: Print the incoming arguments
    println!("Reducer `increment_counter` called with ID: {:?}, Amount: {}",
id, amount);

    let mut counter = Counter::filter_by_id(&id)
        .ok_or_else(|| {
            let error_msg = format!("Counter with id {} not found.", id);
            // Debugging step 2: Log error before returning
            eprintln!("Error in `increment_counter`: {}", error_msg);
            error_msg
        })?;

    // Debugging step 3: Inspect the counter object before modification
    dbg!(&counter); // `dbg!` takes a reference and prints it

    counter.value = counter.value.checked_add(amount)
        .ok_or_else(|| "Counter overflowed!".to_string())?;

    // Debugging step 4: Inspect the counter object after modification
    dbg!(&counter);

    counter.update();

    // Debugging step 5: Confirm success
    println!("Counter with ID {:?} successfully incremented to {}", id,
counter.value);

    Ok(())
}

// ... existing code ...
```

Explanation: * `println!` is used for simple messages, showing the input parameters and success messages. * `eprintln!` is used for error messages, which typically go to standard error, making them distinct in logs. * `dbg!(&counter)` is used to print the `counter` struct's content before and after modification. The `&` means we're passing a reference, so `dbg!` doesn't consume ownership of `counter`.

Now, rebuild and redeploy your module by saving `src/lib.rs`. The `spacetime-cli dev` command should automatically detect changes and recompile.

Let's call these reducers from the `spacetime-cli` to see the logs. Open a new terminal window (keep `spacetime-cli dev` running in the first one).

```
# Create a new counter
spacetime-cli call create_counter --args '[{"id": 123, "initial_value": 0}]'

# Increment the counter
spacetime-cli call increment_counter --args '[{"id": 123, "amount": 5}]'

# Try to increment a non-existent counter (should trigger error log)
spacetime-cli call increment_counter --args '[{"id": 456, "amount": 10}]'
```

Go back to your first terminal running `spacetime-cli dev`. You should see the `println!` and `dbg!` outputs mixed in with the SpaceTimeDB server logs. Notice how `dbg!` includes file, line, and value, which is incredibly helpful!

Step 2: Unit Testing a Reducer

Now, let's write a unit test for our `increment_counter` reducer. SpaceTimeDB reducers are just Rust functions, so we can test them using Rust's standard testing framework.

Create a new file `src/tests.rs` in your project root, or add the tests directly to `src/lib.rs` inside a `#[cfg(test)]` module. For simplicity, let's add it to `src/lib.rs`.

Add the following block to the end of your `src/lib.rs` file:

```
// src/lib.rs (add this at the very end of the file)
#[cfg(test)]
mod tests {
    use super::*;
    use spacetimedb::{
        test_client::{
            ClientDb,
            ReducerContext, // Use the test client's ReducerContext
        },
        table,
    };

    // Define a test-specific Counter table if needed, or reuse the main one
    // For unit tests, we often mock the tables or use the same table
definition
    // but operate on a test-specific ClientDb.

    #[test]
    fn test_create_counter_success() {
        let mut client_db = ClientDb::new();
        let ctx = ReducerContext::new(
            "test_identity".to_string(),
            client_db.timestamp_millis()
        );

        let id = u384::from(100);
        let initial_value = 50;

        // Call the reducer
        let result = create_counter(ctx, id, initial_value);

        // Assert the result is Ok
        assert!(result.is_ok(), "create_counter should succeed");

        // Verify the counter was inserted
        let counter = client_db.get_one::<Counter>().expect("Counter should
exist");
        assert_eq!(counter.id, id);
        assert_eq!(counter.value, initial_value);
    }

    #[test]
    fn test_create_counter_already_exists() {
        let mut client_db = ClientDb::new();
        let ctx = ReducerContext::new(
            "test_identity".to_string(),
            client_db.timestamp_millis()
        );

        let id = u384::from(101);
        let initial_value = 10;

        // First, create the counter
        let _ = create_counter(ctx.clone(), id,
initial_value).unwrap(); // Use clone for ctx

        // Try to create it again
        let result = create_counter(ctx, id, 20);

        // Assert that it returns an error
        assert!(result.is_err(),
```

```
"create_counter should fail if counter already exists");
        assert_eq!(result.unwrap_err(), format!("Counter with id {} already
exists.", id));
    }

    #[test]
    fn test_increment_counter_success() {
        let mut client_db = ClientDb::new();
        let ctx = ReducerContext::new(
            "test_identity".to_string(),
            client_db.timestamp_millis()
        );

        let id = u384::from(1);
        let initial_value = 10;
        let increment_amount = 5;

        // First, create the counter for the test
        client_db.insert(Counter { id, value: initial_value });

        // Call the increment reducer
        let result = increment_counter(ctx, id, increment_amount);

        // Assert the result is Ok
        assert!(result.is_ok(), "increment_counter should succeed");

        // Verify the counter value was updated
        let counter = client_db.get_one::<Counter>().expect("Counter should
exist after increment");
        assert_eq!(counter.id, id);
        assert_eq!(counter.value, initial_value + increment_amount);
    }

    #[test]
    fn test_increment_counter_not_found() {
        let client_db = ClientDb::new(); // No need for mut if we're not
inserting/updating
        let ctx = ReducerContext::new(
            "test_identity".to_string(),
            client_db.timestamp_millis()
        );

        let non_existent_id = u384::from(999);
        let increment_amount = 1;

        // Call the increment reducer on a non-existent ID
        let result = increment_counter(ctx, non_existent_id, increment_amount);

        // Assert that it returns an error
        assert!(result.is_err(), "increment_counter should fail if counter not
found");
        assert_eq!(result.unwrap_err(), format!("Counter with id {} not
found.", non_existent_id));
    }

    #[test]
    fn test_increment_counter_overflow() {
        let mut client_db = ClientDb::new();
        let ctx = ReducerContext::new(
            "test_identity".to_string(),
            client_db.timestamp_millis()
        );
```

```
        let id = u384::from(2);
        let initial_value = u32::MAX - 1; // Close to max value
        let increment_amount = 2; // Will cause overflow

        // Create the counter with a value near max
        client_db.insert(Counter { id, value: initial_value });

        // Call the increment reducer
        let result = increment_counter(ctx, id, increment_amount);

        // Assert that it returns an error
        assert!(result.is_err(), "increment_counter should fail on overflow");
        assert_eq!(result.unwrap_err(), "Counter overflowed!".to_string());
    }
}
```

Explanation: * `#[cfg(test)] mod tests { ... }`: This module will only be compiled when running tests (`cargo test`). * `use spacetimedb::test_client::{ClientDb, ReducerContext};`: We import `ClientDb` which acts as an in-memory database instance for testing, and `ReducerContext` specifically designed for tests. * `#[test]`: Marks a function as a test function. * `ClientDb::new()`: Creates a fresh, empty in-memory database for each test, ensuring isolation. * `client_db.insert(Counter { ... })`: Allows you to set up the initial state of your database for the test. * `client_db.get_one::<Counter>()`: Retrieves a single record from the test database. * `assert!`, `assert_eq!`: Standard Rust macros for asserting conditions.

Now, run your tests from the root of your project:

```
cargo test
```

You should see output indicating that all your tests passed! This demonstrates how you can thoroughly test your reducer logic in isolation, ensuring its determinism and correctness.

Step 3: Client-Side Logging (Conceptual)

While we can't run a full client example here, let's conceptually discuss how you'd enable logging for a TypeScript/JavaScript client.

When initializing your SpaceTimeDB client, you typically have options to configure logging:

```
// Example: src/client/index.ts (conceptual)
import { SpacetimeDBClient } from '@clockworklabs/spacetimedb-client';

// Assume SpaceTimeDB is running locally on default port 3000
const client = new SpacetimeDBClient('ws://localhost:3000');

// Set log level (e.g., 'debug', 'info', 'warn', 'error', 'none')
// The exact method might vary slightly with client library updates,
// but usually involves a config object or a dedicated method.
client.setLogLevel('debug'); // Or client.configure({ logLevel: 'debug' });

client.onConnect(() => {
    console.log('SpacetimeDB client connected!');
    // Example: subscribe to the Counter table
    client.subscribe(['Counter']);
});

client.onDisconnect(() => {
    console.warn('SpacetimeDB client disconnected!');
});

client.on('error', (error) => {
    console.error('SpacetimeDB client error:', error);
});

// Listen for table updates
client.on('Counter:insert', (counter) => {
    console.log('New Counter inserted:', counter);
});

client.on('Counter:update', (oldValue, newValue) => {
    console.log('Counter updated from', oldValue, 'to', newValue);
});

client.connect();

// Later, call a reducer
// client.call('create_counter', 123, 0);
// client.call('increment_counter', 123, 1);
```

By enabling `debug` level logging, your client's console would show detailed messages about WebSocket frames, subscription states, and reducer call acknowledgements. This is invaluable for understanding client-side synchronization issues.

Mini-Challenge: Test a Conditional Reducer

It's your turn! Let's add a new reducer and write tests for it.

Challenge: Create a new reducer called `decrement_counter` that takes an `id` and an `amount`. This reducer should: 1. Find the `Counter` by `id`. If not found, return an error. 2. Decrement its `value` by `amount`. 3. **Crucially:** If decrementing

would make the `value` go below zero, it should instead return an error ("Cannot decrement below zero!"). 4. Update the `Counter` in the database.

After implementing the reducer, write at least **three unit tests** for `decrement_counter` in your `src/lib.rs`'s `#[cfg(test)]` block: * One test for a successful decrement. * One test for decrementing a non-existent counter. * One test for attempting to decrement below zero.

Hint: * Use `checked_sub` for safe subtraction, similar to how `checked_add` was used. * Remember to set up the `ClientDb` with appropriate initial state for each test.

What to observe/learn: * How to handle conditional logic and error cases within reducers. * How to write comprehensive unit tests that cover different scenarios, including edge cases. * The importance of `Result<(), String>` for reducer return types to convey success or specific errors.

```
// Add this new reducer to src/lib.rs
#[spacetimedb(reducer)]
pub fn decrement_counter(ctx: ReducerContext, id: u384, amount: u32) ->
Result<(), String> {
    // TODO: Implement the logic here
    unimplemented!("Decrement counter reducer not yet implemented for the
challenge!")
}

// Add your tests within the existing #[cfg(test)] mod tests { ... } block
// For example:
/*
    #[test]
    fn test_decrement_counter_success() {
        // ... your test code ...
    }

    #[test]
    fn test_decrement_counter_not_found() {
        // ... your test code ...
    }

    #[test]
    fn test_decrement_counter_below_zero() {
        // ... your test code ...
    }
*/
```

Once you've implemented the reducer and tests, run `cargo test` to verify your solution.

Common Pitfalls & Troubleshooting

Even with good practices, issues can arise. Knowing common pitfalls helps in quicker diagnosis.

1. **Reducer Non-Determinism**: This is a critical error in SpaceTimeDB. A reducer is non-deterministic if, given the same initial state and input arguments, it can produce different outputs (different final states or different errors).

- **Cause**: Using `ctx.timestamp_millis()` or `ctx.caller()` directly to generate unique IDs or random numbers, or relying on external non-deterministic factors.
- **Troubleshooting**: SpaceTimeDB's runtime is designed to detect non-determinism. If you encounter errors related to "non-deterministic reducer," carefully review your reducer logic.
- **Solution**: For unique IDs, use `id: u384` as a reducer argument, generated client-side, or use the `ctx.timestamp_millis()` for ordering events, but not for generating primary keys directly if multiple clients could call it simultaneously. For randomness, seed your random number generator with a deterministic value (e.g., `id` or `ctx.timestamp_millis()`) and acknowledge that this is pseudo-randomness relative to the deterministic seed. 2. **Stale Client State / Synchronization Issues**: Your client isn't reflecting the latest data from the server.
- **Cause**: * Client not subscribed to the relevant tables. * Network connectivity issues (WebSocket drops). * Client-side caching logic interfering with SpaceTimeDB updates. * Reducer logic not actually updating the expected table.
- **Troubleshooting**:
- **Client-side**: Use browser dev tools (Network tab) to inspect WebSocket traffic. Look for `table_update` messages. Check if your client's `on` listeners are correctly set up and processing updates.
- **Server-side**: Verify that your reducer successfully updates the table using `spacetime-cli logs` or by querying the database directly after a reducer call. 3. **Performance Bottlenecks**: Reducers or queries taking too long, leading to slow updates or a sluggish UI.
- **Cause**: * Inefficient reducer logic (e.g., iterating over large tables, complex calculations). * Schema design issues (missing indexes, overly complex

relations). * Large number of active subscriptions overwhelming the server or client.

- **Troubleshooting**:
- **Server-side**: Use `spacetime-cli logs` to see reducer execution times (SpaceTimeDB often logs these by default). Add `println!`/`dbg!` to time critical sections of your reducer. Ensure appropriate primary keys and indexes are defined in your schema.
- **Client-side**: Monitor client-side frame rates and network latency. Optimize how your client processes updates and renders data. 4. **Reducer Errors Not Propagating**: A reducer fails, but the client doesn't get an error response.
- **Cause**: The client code isn't handling the `Result` of the reducer call correctly, or the error is being swallowed somewhere.
- **Troubleshooting**: Ensure your client-side `client.call()` handler explicitly checks for and logs errors. Review the `spacetime-cli logs` to confirm the reducer actually returned an `Err` and wasn't just silently failing.

Summary

Congratulations! You've reached the end of this crucial chapter on debugging, testing, and observability in SpaceTimeDB. We've covered a lot of ground, equipping you with the knowledge to build more robust and maintainable real-time applications.

Here are the key takeaways:

- **Debugging is essential**: Use `println!`, `eprintln!`, `dbg!`, and Rust debugger integration for server-side (reducer) issues, and browser dev tools with client-side logging for frontend problems.
- **Testing ensures correctness**: SpaceTimeDB's deterministic nature makes it highly testable. Implement unit tests for individual reducers using `cargo test` and `ClientDb` to ensure their logic is sound and robust against edge cases like non-existent data or overflows.
- **Observability is paramount for production**: Beyond development debugging, establish a system for production monitoring using structured logs, comprehensive metrics (reducer latency, error rates, connection counts), and potentially distributed tracing.

- **Common pitfalls**: Be aware of reducer non-determinism, stale client state, performance bottlenecks, and unhandled reducer errors, and know how to troubleshoot them effectively.
- **Version Focus**: We've used SpaceTimeDB v2.x and current Rust practices, ensuring modern best practices.

By integrating these practices into your development workflow, you'll gain confidence in deploying and operating your SpaceTimeDB applications, knowing you have the tools to understand, verify, and diagnose their behavior.

What's Next?

In the next chapter, we'll shift our focus to deployment strategies and scaling considerations for SpaceTimeDB applications, taking your projects from local development to production-ready systems.

References

- SpacetimeDB Official Documentation
- The Rust Programming Language Book
- Rust `log` crate documentation
- Rust `dbg!` macro documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Chapter 16: Schema Evolution, Migrations, and Advanced Design Patterns

Introduction

Welcome to Chapter 16! So far, we've explored the core concepts of SpaceTimeDB, built real-time applications, and even delved into performance and security. But what happens when your application grows, and your initial data model no longer fits your evolving needs? This is where **schema evolution** and **data migrations** come into play.

In this chapter, we'll tackle the crucial, yet often overlooked, aspects of managing change in your SpaceTimeDB projects. We'll learn how to gracefully adapt your database schema over time without disrupting existing data or live applications. We'll also explore different strategies for migrating data when your schema changes require transforming existing information. Finally, we'll dive into advanced design patterns like Event Sourcing and CQRS, showing how SpaceTimeDB's unique architecture naturally supports them, helping you build even more robust and scalable systems.

This chapter is designed for developers who are ready to build production-grade SpaceTimeDB applications that can adapt and grow. We'll assume you're comfortable with defining tables, writing reducers, and interacting with SpaceTimeDB from a client. Let's get ready to master change!

Core Concepts: Schema Evolution in SpaceTimeDB

Schema evolution refers to the process of adapting your database structure over the lifespan of an application. As features are added or business requirements change, your data model will inevitably need adjustments.

What Makes SpaceTimeDB Schema Evolution Unique?

In traditional relational databases (like PostgreSQL or MySQL), schema is defined using Data Definition Language (DDL) statements (e.g., `CREATE TABLE`, `ALTER TABLE`). You often manage these changes with external migration tools (like Flyway or Alembic).

SpaceTimeDB, however, treats your schema as part of your application's module code. Your tables, their fields, and their types are declared directly within your Rust modules. This tight integration means:

1. **Source of Truth:** Your Rust module code is the definitive source of truth for your database schema.
2. **Atomic Deployment:** When you deploy a new version of your SpaceTimeDB module, both the application logic (reducers) and the schema definitions are updated together.
3. **Strong Typing:** Rust's strong type system provides compile-time checks, catching many potential schema-related errors before deployment.

The Challenge of Change

Imagine you have a `Player` table:

```
// modules/game/src/lib.rs
#[spacetimedb(table)]
pub struct Player {
    #[primarykey]
    pub id: u64,
    pub username: String,
    pub score: u32,
}
```

Now, your game needs to store a player's `level` and a `last_login` timestamp. How do you add these fields without losing existing player data? What if you later decide to rename `username` to `display_name`? These are the challenges of schema evolution.

SpaceTimeDB's Approach to Module Updates

When you deploy a new version of your SpaceTimeDB module, the system intelligently compares the new schema definition with the existing one.

- **Adding New Fields:** If you add a new field to an existing table, SpaceTimeDB will add that column. For existing rows, the new field will typically be initialized with its default value (e.g., `0` for numbers, empty string for `String`, `None` for `Option<T>`). This is usually a backward-compatible change.
- **Removing Fields:** If you remove a field, SpaceTimeDB will remove the corresponding column. This is a breaking change for any clients or reducers that rely on that field. The data in that column will be lost.

- **Changing Field Types:** Changing a field's type (e.g., `u32` to `String`) is almost always a breaking change and will likely result in data loss or conversion errors for existing data.
- **Renaming Fields/Tables:** This is essentially a removal of the old and addition of the new. It's a breaking change.

Key takeaway: While SpaceTimeDB manages the underlying database changes, you are responsible for ensuring that your application logic (reducers, client code) can handle these schema changes, especially regarding existing data.

Backward and Forward Compatibility

- **Backward Compatibility:** A new version of your schema is backward-compatible if old clients or reducers can still interact with it without breaking. Adding nullable fields is often backward-compatible.
- **Forward Compatibility:** An old client/reducer can interact with a newer schema. This is harder to achieve and less common, as older code typically doesn't know about new fields.

For SpaceTimeDB, the focus is primarily on **backward compatibility** for your clients and ensuring your reducers gracefully handle schema changes.

Core Concepts: Data Migrations

Sometimes, evolving your schema isn't enough; you need to transform your existing data to fit the new schema. This is where data migrations come in.

Why Data Migrations?

Consider these scenarios:

- **Splitting a Field:** You have a `full_name: String` field and decide to split it into `first_name: String` and `last_name: String`. Existing `full_name` data needs to be parsed and moved.
- **Combining Fields:** You have `address_line1`, `address_line2`, `city`, `zip_code` and decide to consolidate into a single `address_json: String`.
- **Data Normalization/Denormalization:** Changing how data is structured across tables.
- **Populating New Fields:** As in our `Player` example, when adding `display_name`, you might want to pre-populate it for existing players based on their `username`.

SpaceTimeDB offers a few strategies for data migrations:

1. In-Reducer Migrations (Limited Scope)

For very simple migrations, you can embed logic directly within your reducers to handle older data formats.

How it works: When a reducer reads data, it checks if a field exists or is in an old format. If so, it performs a conversion on the fly and then writes the updated data back.

Pros: * No separate migration script needed. * Data is updated opportunistically as it's accessed.

Cons: * Can make reducers more complex and harder to read. * Doesn't "clean up" all old data at once; only updates when a row is touched. * Not suitable for large-scale transformations or changes that affect many rows infrequently.

2. One-Off Module-Based Migrations

This is a common and robust approach in SpaceTimeDB. You create a temporary reducer or a specific module function designed solely to perform the data migration.

How it works: 1. Define a new reducer (e.g., `migrate_player_data`). 2. Inside this reducer, iterate through the relevant table(s). 3. For each row, apply your data transformation logic. 4. Update the row using `SpaceTimeDB.update()`. 5. Crucially, add a mechanism to ensure this reducer runs only once (e.g., a special "migration_status" table, or a check that it has already run). 6. Deploy your module. 7. Call the migration reducer from a client. 8. Once successfully run and verified, you can remove the migration reducer from your module in a subsequent deployment.

Pros: * Keeps migration logic separate from core application reducers. * Ensures all relevant data is transformed consistently. * Leverages SpaceTimeDB's transactional and real-time update guarantees.

Cons: * Requires careful management to ensure it runs only once. * Can be resource-intensive for very large datasets, potentially impacting live system performance.

3. External Script Migrations (for Complex Scenarios)

For highly complex, large-scale, or highly customized data transformations, you might opt for an external script.

How it works: 1. Write a script (e.g., in TypeScript/Node.js or Python) that connects to your SpaceTimeDB instance using its client SDK. 2. The script queries data, performs transformations in memory, and then sends update commands back to SpaceTimeDB. 3. This script often bypasses reducers directly, using privileged client connections if necessary, or calling specific reducers designed for raw data updates.

Pros: * Full control over the migration process. * Can leverage external libraries and tooling for complex data processing. * Can be run offline or during maintenance windows.

Cons: * Bypasses reducer logic, potentially violating business rules if not carefully managed. * Requires careful transaction management and error handling in the external script. * Potential for race conditions if the system is live and other clients are modifying data. * Less integrated with SpaceTimeDB's core consistency model. **Use with extreme caution and only when necessary.**

Hands-on: Implementing a Simple Schema Evolution and Migration

Let's put these concepts into practice. We'll simulate a game scenario where we need to add a `display_name` field to our `Player` table and populate it for existing players.

Scenario: Adding and Populating `display_name`

Imagine our game has been running for a while. Players have `id`, `username`, and `score`. We now want to introduce a `display_name` field, which should initially be the same as `username` for existing players, but can be changed later.

Step 1: Initial Schema (Recap)

First, let's establish our initial `Player` table in `modules/game/src/lib.rs`.

```
// modules/game/src/lib.rs
// (Make sure you have your standard SpaceTimeDB module boilerplate)

use spacetimedb::{spacetimedb, table, ReducerContext, Identity, timestamp};

#[spacetimedb(table)]
pub struct Player {
    #[primarykey]
    pub id: u64,
    pub username: String,
    pub score: u32,
}

// A simple reducer to create a player for testing
#[spacetimedb(reducer)]
pub fn create_player(ctx: ReducerContext, username: String) {
    let id = Player::iter().count() as u64 + 1; // Simple ID generation
    Player::insert(Player {
        id,
        username,
        score: 0,
    }).expect("Failed to insert player");
    log::info!("Player created: {}", username);
}

// ... other reducers or event definitions as needed ...
```

Action: 1. Save the above code in `modules/game/src/lib.rs`. 2. Deploy this initial version: `bash spacetime deploy` 3. Connect a client (e.g., your frontend or a simple Node.js script) and create a few players: ```javascript // Example client-side code (Node.js or browser) import { SpacetimeDBClient } from "@clockworklabs/spacetimedb-sdk"; import { createPlayer } from "./module_bindings"; // Assuming generated bindings

```
const client = new SpacetimeDBClient("ws://localhost:3000"); // Adjust if not
local
client.connect();

client.onConnect(() => {
    console.log("Connected to SpaceTimeDB!");
    createPlayer("Alice");
    createPlayer("Bob");
    createPlayer("Charlie");
});

// You can also subscribe to the Player table to see data
client.subscribe([{tableName: "Player"}]);
client.on("Player:insert", (player) => console.log("New Player:", player));
```
Run this client code and ensure Alice, Bob, and Charlie are created. You can verify with `spacetime client` or `spacetime db dump`.

Step 2: Evolving the Schema

Now, let's add the `display_name` field to our `Player` table. We'll make it an `Option<String>` initially to handle cases where it might not be immediately set, but for our migration, we'll populate it.

Modify `modules/game/src/lib.rs`:

```
// modules/game/src/lib.rs
// ... (existing use statements) ...

#[spacetimedb(table)]
pub struct Player {
    #[primarykey]
    pub id: u664, // Corrected u64 typo from previous thought process
    pub username: String,
    pub score: u32,
    pub display_name: Option<String>, // <--- NEW FIELD!
}

// ... (existing create_player reducer) ...
```

Explanation: We've added `pub display_name: Option<String>,`. When you deploy this change, SpaceTimeDB will add a new `display_name` column to the `Player` table. For all existing rows (Alice, Bob, Charlie), this new field will be `None`. New players created after this deployment would also have `display_name` as `None` by default unless explicitly set in `create_player` or another reducer.

Action: 1. Update `modules/game/src/lib.rs` with the `display_name` field. 2. Deploy the updated module: `bash spacetime deploy` You should see a message indicating schema changes were applied. 3. Verify the schema change (e.g., `spacetime db dump Player` or connect with `spacetime client` and inspect the schema). You'll see `display_name` as `null` for existing players.

Step 3: Creating a Migration Reducer

Now, let's write a one-off migration reducer to populate `display_name` for existing players. We'll also add a simple `MigrationStatus` table to ensure this migration runs only once.

Add the following to `modules/game/src/lib.rs`:

```rust
// modules/game/src/lib.rs
// ... (existing Player table and create_player reducer) ...

#[spacetimedb(table)]
pub struct MigrationStatus {
    #[primarykey]
    pub migration_name: String,
    pub completed_at: u64, // Timestamp
}

#[spacetimedb(reducer)]
pub fn migrate_player_display_names(_ctx: ReducerContext) {
    let migration_name = "populate_player_display_names".to_string();

    // Check if migration has already been completed
    if MigrationStatus::filter_by_migration_name(&migration_name).is_some() {
        log::info!("Migration '{}' already completed. Skipping.", migration_nam
e);
        return;
    }

    log::info!("Starting migration: '{}'", migration_name);

    // Iterate through all players and update their display_name
    for mut player in Player::iter() {
        if player.display_name.is_none() {
            // Only update if display_name is not already set
            player.display_name = Some(player.username.clone());
            Player::update(player.id, player).expect("Failed to update player
during migration");
            log::info!("Updated player {} (id: {}) with display_name: {}", play
er.username, player.id, player.display_name.as_ref().unwrap());
        }
    }

    // Mark migration as completed
    MigrationStatus::insert(MigrationStatus {
        migration_name,
        completed_at: timestamp(),
    }).expect("Failed to insert migration status");

    log::info!("Migration completed successfully!");
}
```

Explanation: 1. `MigrationStatus` **Table:** This new table is a simple way to track which migrations have run. It has a `migration_name` (primary key) and `completed_at` timestamp. 2. `migrate_player_display_names` **Reducer:** * It defines a `migration_name` constant. * It first checks `MigrationStatus::filter_by_migration_name` to see if this migration has already been recorded as completed. If so, it exits early, preventing re-execution (this is crucial for **idempotency**). * It then iterates through `Player::iter()`, which gives us mutable references to each `Player` row. * Inside the loop, it checks if `player.display_name` is `None`. If it is, we populate it with the `player.username`. * `Player::update(player.id, player)` commits the

changes to the database. * After iterating through all players, it inserts a record into `MigrationStatus` to mark the migration as done.

Step 4: Deploying and Executing the Migration

Now, deploy the module with the new `MigrationStatus` table and the migration reducer. Then, call the reducer from your client.

Action: 1. Add the `MigrationStatus` table and `migrate_player_display_names` reducer to `modules/game/src/lib.rs`. 2. Deploy the module: `bash spacetime deploy` 3. From your client-side code, call the migration reducer. Ensure you have the `migratePlayerDisplayNames` binding generated.

```
```javascript
// Example client-side code (Node.js or browser)
import { SpacetimeDBClient } from "@clockworklabs/spacetimedb-sdk";
import { createPlayer, migratePlayerDisplayNames } from "./module_bindings"; //
Assuming generated bindings

const client = new SpacetimeDBClient("ws://localhost:3000");
client.connect();

client.onConnect(() => {
    console.log("Connected to SpaceTimeDB!");

    // Call the migration reducer
    migratePlayerDisplayNames();

    // Optionally, create a new player to see how display_name behaves
    // createPlayer("David");
});

client.subscribe([{ tableName: "Player" }, { tableName: "MigrationStatus" }]);
client.on("Player:insert", (player) => console.log("New Player:", player));
client.on("Player:update", (oldPlayer, newPlayer) => console.log("Player
Updated:", newPlayer));
client.on("MigrationStatus:insert", (status) => console.log("Migration
Status:", status));
```
```

1. Run the client. You should see log messages indicating the migration starting, players being updated, and the migration completing.
2. Verify the data:
    - Use `spacetime db dump Player` to confirm `display_name` is now set for Alice, Bob, and Charlie.
    - Use `spacetime db dump MigrationStatus` to confirm the `populate_player_display_names` migration is recorded.

Step 5: Cleaning Up (Optional but Recommended)

Once the migration has successfully run and you've verified the data, you can (and often should) remove the `migrate_player_display_names` reducer and potentially the `MigrationStatus` table from your module. This keeps your production module lean and prevents accidental re-execution.

Action: 1. Comment out or delete the `MigrationStatus` table and `migrate_player_display_names` reducer from `modules/game/src/lib.rs`. 2. Deploy the module again: `bash spacetime deploy` SpaceTimeDB will detect that `MigrationStatus` and the reducer are no longer defined and will remove them. The data in `Player` will remain as migrated.

This hands-on exercise demonstrates a practical way to evolve your schema and migrate data using SpaceTimeDB's module-based approach.

Advanced Design Patterns for SpaceTimeDB

SpaceTimeDB's unique architecture – combining a database, real-time synchronization, and deterministic server-side logic – makes it an excellent fit for several advanced architectural patterns.

1. Event Sourcing with SpaceTimeDB

What is Event Sourcing? Instead of storing the current state of an application, Event Sourcing stores every change to the state as a sequence of immutable events. The current state is then derived by replaying these events.

How SpaceTimeDB Supports It: SpaceTimeDB's reducer model is inherently event-driven. Each reducer call is essentially an "event" that modifies the global state. You can extend this by:

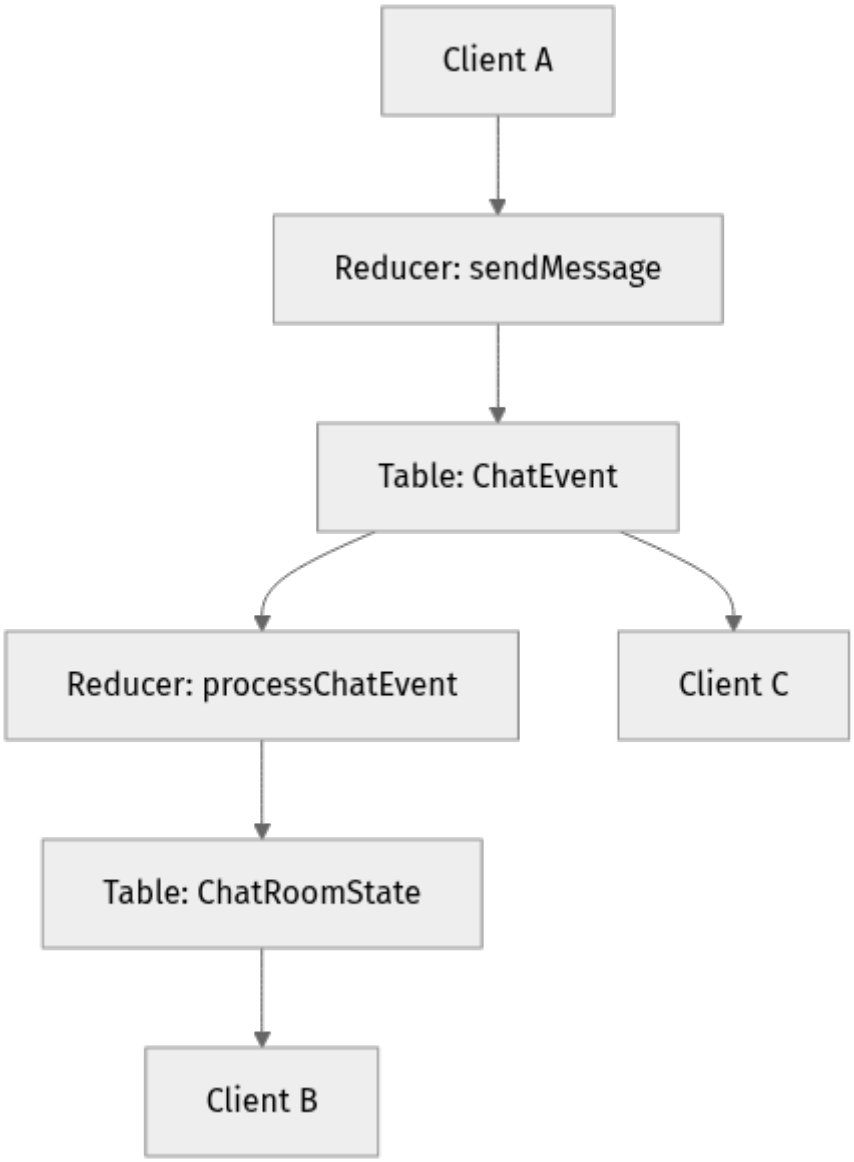
- **Event Tables:** Create dedicated tables to store historical events. For example, a `ChatEvent` table might store `MessageSent`, `UserJoined`, `UserLeft` events.
- **State Derivation:** Your main application tables (e.g., `ChatRoomState`) can then be populated or updated by reducers that process these events.

Benefits:

- **Auditability:** A complete, immutable history of every change.
- **Time Travel:** Replay events to reconstruct state at any point in time.
- **Debugging:** Easier to trace how a particular state was reached.
- **Consistency:** Events are processed deterministically by reducers.

Example: Simple Chat Application

Let's imagine a `ChatRoom` with messages.



Mermaid Diagram: Event Sourcing Flow with SpaceTimeDB

In this pattern: 1. A client calls a reducer (e.g., `send_message`). 2. This reducer first records a new `MessageSent` event into a `ChatEvent` table. 3. Then, another internal reducer (or the same one) processes this `ChatEvent` to update the `ChatRoomState` table, which holds the current, derived list of messages. 4. Clients subscribe to `ChatRoomState` to see the live chat, and potentially `ChatEvent` for an audit log.

2. Command-Query Responsibility Segregation (CQRS)

What is CQRS? CQRS separates the concerns of modifying data ("Commands") from reading data ("Queries"). This means you might have different models or even different databases optimized for writes versus reads.

How SpaceTimeDB Supports It: SpaceTimeDB naturally aligns with CQRS:

- **Commands (Writes):** Your reducers are the "command handlers." They take input (commands), apply business logic, and modify the database state. This is your write model.
- **Queries (Reads):** Client subscriptions to tables are your "query model." Clients subscribe to the data they need, and SpaceTimeDB streams real-time updates. This is your read model.

Benefits:

- **Scalability:** Read and write paths can be optimized and scaled independently.
- **Flexibility:** Read models can be denormalized for optimal querying without affecting the write model's integrity.
- **Performance:** Queries can be highly optimized for specific client needs.

Example: User Profile Management

You might have a `User` table for core user data (write model, updated by reducers) and a `UserProfileView` table (a derived, denormalized table updated by other reducers) that combines `User` data with `GameStats` for quick display on a profile page (read model).

3. Distributed Counters and Aggregates

Challenge: How do you efficiently manage shared counters (e.g., "total active users," "likes on a post") or aggregates (e.g., "sum of all player scores") in a real-time, distributed system?

SpaceTimeDB Solution: Reducers are perfect for this. Because reducers execute deterministically and atomically, they can safely increment/decrement counters or update aggregates without race conditions.

```rust
#[spacetimedb(table)]
pub struct GlobalStats {
    #[primarykey]
    pub key: String, // e.g., "total_users", "total_posts"
    pub value: u64,
}

#[spacetimedb(reducer)]
pub fn increment_user_count(_ctx: ReducerContext) {
    let key = "total_users".to_string();
    let mut stat = GlobalStats::filter_by_key(&key)
        .unwrap_or_else(|| GlobalStats { key: key.clone(), value: 0 });
    stat.value += 1;
    GlobalStats::update_or_insert(stat);
}
```

This reducer ensures that `total_users` is incremented safely and atomically, even with many concurrent calls.

4. State Machines

What is a State Machine? A state machine models the behavior of an entity by representing its possible states and the transitions between them. For example, an order might go from `Pending` -> `Processing` -> `Shipped` -> `Delivered`.

How SpaceTimeDB Supports It: SpaceTimeDB tables can store the current state of an entity, and reducers can enforce the valid transitions.

```rust
// Example: Order Status
pub enum OrderStatus {
    Pending,
    Processing,
    Shipped,
    Delivered,
    Cancelled,
}

#[spacetimedb(table)]
pub struct Order {
    #[primarykey]
    pub id: u64,
    pub customer_id: u64,
    pub status: OrderStatus,
    // ... other order details
}

#[spacetimedb(reducer)]
pub fn ship_order(_ctx: ReducerContext, order_id: u64) {
    let mut order = Order::filter_by_id(&order_id)
        .expect("Order not found.");

    match order.status {
        OrderStatus::Processing => {
            order.status = OrderStatus::Shipped;
            Order::update(order_id, order).expect("Failed to update order
status.");
            log::info!("Order {} shipped.", order_id);
        }
        _ => {
            log::warn!("Cannot ship order {} from status: {:?}", order_id, orde
r.status);
            // You might want to return an error or revert transaction in a
real system
        }
    }
}
```

This reducer ensures that an order can only be `Shipped` if its current status is `Processing`, enforcing the state machine's rules.

Mini-Challenge: Implement an Event-Sourced Like Feature

Let's enhance our simple game (or a new project if you prefer) to incorporate an event-sourced pattern for player actions.

Challenge: 1. Create a new table called `PlayerActionEvent`. This table should record events like `PlayerMoved` or `ItemPickedUp`. It should store: * An `id` (primary key, `u64`). * `player_id` (`u64`). * `event_type` (`String`, e.g., "Moved", "PickedUp"). * `details` (`String`, a JSON string or simple description, e.g., "Moved to (10, 5)", "Picked up 'Health Potion'"). * `timestamp` (`u64`). 2. Modify an

existing reducer (or create a new one) that handles a player action (e.g., `move_player` or `pick_up_item`). In addition to updating the player's position or inventory, this reducer should also insert a new row into the `PlayerActionEvent` table. 3. Create a new derived table, `PlayerRecentActions`, which stores the last 3 actions for each player. This table should be updated by another reducer that listens to `PlayerActionEvent` inserts. (Hint: This will involve deleting old actions if a player has more than 3.) 4. From your client, call the action reducer a few times for a player and then subscribe to `PlayerRecentActions` to observe the derived state.

Hint: * For `PlayerActionEvent`, `details` could be an `Option<String>` if some events have no details. * For `PlayerRecentActions`, you might need a composite primary key or a unique index on `(player_id, action_timestamp)` to manage ordering, or simply filter and sort within your reducer. * Remember `timestamp()` for recording event times.

What to observe/learn: * How to combine direct state updates with event logging. * The concept of creating "event streams" in SpaceTimeDB. * How to derive and maintain a separate, optimized read model (like `PlayerRecentActions`) from an event stream using reducers. * The power of SpaceTimeDB's deterministic execution for building consistent derived views.

Common Pitfalls & Troubleshooting

1. Breaking Schema Changes Without Planning

- **Pitfall:** Deploying a schema change (e.g., removing a field, changing a type) without considering its impact on existing data or client applications.
- **Troubleshooting:**
  - **Always test schema changes in a development environment first.**
  - For breaking changes, plan a multi-step deployment:
    1. Add new fields (make them optional or default initially).
    2. Migrate data to the new fields.
    3. Update clients/reducers to use new fields.
    4. (Optional) Remove old fields.
  - Use `Option<T>` for new fields to ensure backward compatibility during transitions.

2. Non-Idempotent Migrations

- **Pitfall:** A migration reducer that, if run multiple times, would cause incorrect data (e.g., incrementing a value multiple times when it should only be incremented once).
- **Troubleshooting:**
  - **Always design migrations to be idempotent.** This means running them once or a hundred times yields the same correct result.
  - Our `MigrationStatus` table pattern is a good example of ensuring idempotency by checking if a migration has already run.
  - For updates, check the current state before applying a change (e.g., `if player.display_name.is_none() { ... }`).

3. Over-Complicating In-Reducer Migrations

- **Pitfall:** Embedding too much complex data transformation logic directly into core application reducers.
- **Troubleshooting:**
  - Keep core reducers focused on their primary business logic.
  - If a migration is complex or affects many rows, use the "one-off module-based migration" strategy to isolate the logic.
  - Long-running reducers can impact performance; separate them if they need to iterate over large datasets.

4. Race Conditions with External Migrations

- **Pitfall:** Running an external script that directly modifies SpaceTimeDB data while other clients are actively writing, leading to inconsistent states.
- **Troubleshooting:**
  - **Avoid external script migrations unless absolutely necessary.** Prefer module-based reducers.
  - If you must use an external script, run it during a maintenance window when no other writes are occurring.
  - Consider implementing locking mechanisms (e.g., a "migration_in_progress" flag in a SpaceTimeDB table) that reducers can check before processing updates, though this adds complexity.

Summary

Phew! You've just mastered some of the most advanced concepts in SpaceTimeDB development. Let's recap the key takeaways from this chapter:

- **Schema Evolution:** SpaceTimeDB's schema is defined in your Rust modules, providing a unified source of truth. Changes are applied atomically on deployment.
- **Managing Change:** Understand the implications of adding, removing, or changing field types. Prioritize backward compatibility for your clients and reducers.
- **Data Migrations:**
- **In-Reducer:** For simple, opportunistic data fixes.
- **One-Off Module-Based:** The recommended approach for structured migrations, using a temporary reducer to transform data and a `MigrationStatus` table for idempotency.
- **External Script:** For highly complex scenarios, but use with extreme caution due to potential consistency issues.
- **Advanced Design Patterns:**
- **Event Sourcing:** Naturally supported by SpaceTimeDB's reducer model, allowing you to store immutable event streams and derive application state.
- **CQRS:** SpaceTimeDB inherently separates command (reducer) and query (subscription) responsibilities, enabling optimized read/write paths.
- **Distributed Counters:** Reducers provide a safe and atomic way to manage shared counters and aggregates.
- **State Machines:** Model entity lifecycles by storing state in tables and enforcing transitions within deterministic reducers.
- **Best Practices:** Always test schema changes, ensure migrations are idempotent, and choose the right migration strategy for the complexity of the task.

You now have the tools and knowledge to not only build powerful real-time applications with SpaceTimeDB but also to maintain and evolve them gracefully over time. This is a critical skill for any long-lived project.

In the next chapter, we'll shift our focus to deployment strategies, ensuring your finely crafted SpaceTimeDB application can run reliably in production environments.

References

- SpacetimeDB Official Documentation
- SpacetimeDB GitHub Repository
- Rust Programming Language Documentation
- Event Sourcing Pattern (Microsoft Docs)
- CQRS Pattern (Martin Fowler)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Chapter 17: Production Best Practices: From Development to Deployment

Chapter 17: Production Best Practices: From Development to Deployment

Welcome back, intrepid SpaceTimeDB architect! You've come a long way, learning how to build powerful, real-time applications, design schemas, write efficient reducers, and handle client synchronization. So far, our focus has largely been on the "development" aspect—getting things working. But what happens when your amazing multiplayer game or collaborative app is ready for the world? That's where production best practices come in!

This chapter is your guide to transitioning your SpaceTimeDB application from a local development environment to a robust, scalable, and secure production deployment. We'll cover essential topics like environment configuration, deployment strategies, how to monitor your application in the wild, and crucial security considerations. By the end of this chapter, you'll have a solid understanding of what it takes to confidently launch and maintain your SpaceTimeDB-powered systems, ensuring they're ready for prime time.

Ready to make your SpaceTimeDB project shine in production? Let's dive in!

Core Concepts for Production Readiness

Moving to production means thinking beyond just "does it work?" to "does it work reliably, securely, and efficiently for everyone, all the time?" This requires a shift in mindset and a focus on several key areas.

1. Environment Configuration: Keeping Dev and Prod Separate

One of the first things you'll encounter is the need for different settings between your development environment and your production environment. For example, you might want verbose logging during development but only critical errors in production. Database connection strings, API keys, and external service URLs will definitely differ.

What is it? Environment configuration refers to managing settings and parameters that change based on the deployment environment (development, staging, production, etc.). **Why is it important?**

- **Security:** Prevents sensitive production credentials from being exposed in development code.
- **Flexibility:** Allows you to tune application behavior (e.g., logging verbosity, performance settings) for each environment.
- **Reliability:** Ensures your application connects to the correct services and databases. **How it functions:** The most common and secure way to handle environment-specific configurations is by using **environment variables**. Your application reads these variables at runtime, adapting its behavior accordingly.

SpaceTimeDB modules (written in Rust) can access environment variables using standard Rust libraries. For example, `std::env::var("MY_SETTING")`.

2. Deployment Strategies: Getting Your Code to the Cloud

Once your SpaceTimeDB module and client application are ready, you need a way to package and deploy them.

A. Containerization with Docker

Containerization has become the de-facto standard for deploying modern applications. Docker allows you to package your application and all its dependencies into a single, isolated unit called a container.

What is it? Docker packages your application, its dependencies, and its configuration into a portable image. This image can then be run as a container on any system that has Docker installed. **Why is it important?**

- **Consistency:** "Works on my machine" becomes "works everywhere" because the environment is standardized.
- **Isolation:** Containers run in isolation from each other and the host system.
- **Portability:** Easily move your application between different environments (local, staging, production). **How it functions:** You define a `Dockerfile` that specifies how to build your application's image. This includes things like the base operating system, installing dependencies, copying your code, and defining the command to run your application.

For SpaceTimeDB, you would containerize your compiled SpaceTimeDB module and client applications separately. The SpaceTimeDB server itself can also be run in a container.

B. Orchestration with Kubernetes (Briefly)

For complex, large-scale deployments, especially those involving multiple services and high availability, **Kubernetes** is the industry leader for container orchestration. It automates the deployment, scaling, and management of containerized applications. While a deep dive into Kubernetes is beyond this chapter, understand that if your SpaceTimeDB application grows significantly, you'll likely deploy it to a Kubernetes cluster.

C. Continuous Integration/Continuous Deployment (CI/CD)

CI/CD pipelines automate the process of building, testing, and deploying your application.

What is it?

- **Continuous Integration (CI):** Developers frequently merge their code changes into a central repository. Automated builds and tests run to detect integration issues early.
- **Continuous Deployment (CD):** After successful CI, changes are automatically deployed to production. **Why is it important?**
- **Speed:** Faster release cycles.
- **Reliability:** Automated tests catch bugs before deployment.
- **Consistency:** Standardized deployment process. **How it functions:** Tools like GitHub Actions, GitLab CI/CD, Jenkins, or CircleCI monitor your code repository. When changes are pushed, they trigger a pipeline that compiles your SpaceTimeDB module, runs tests, builds Docker images, and pushes them to a container registry, eventually deploying them to your chosen environment.

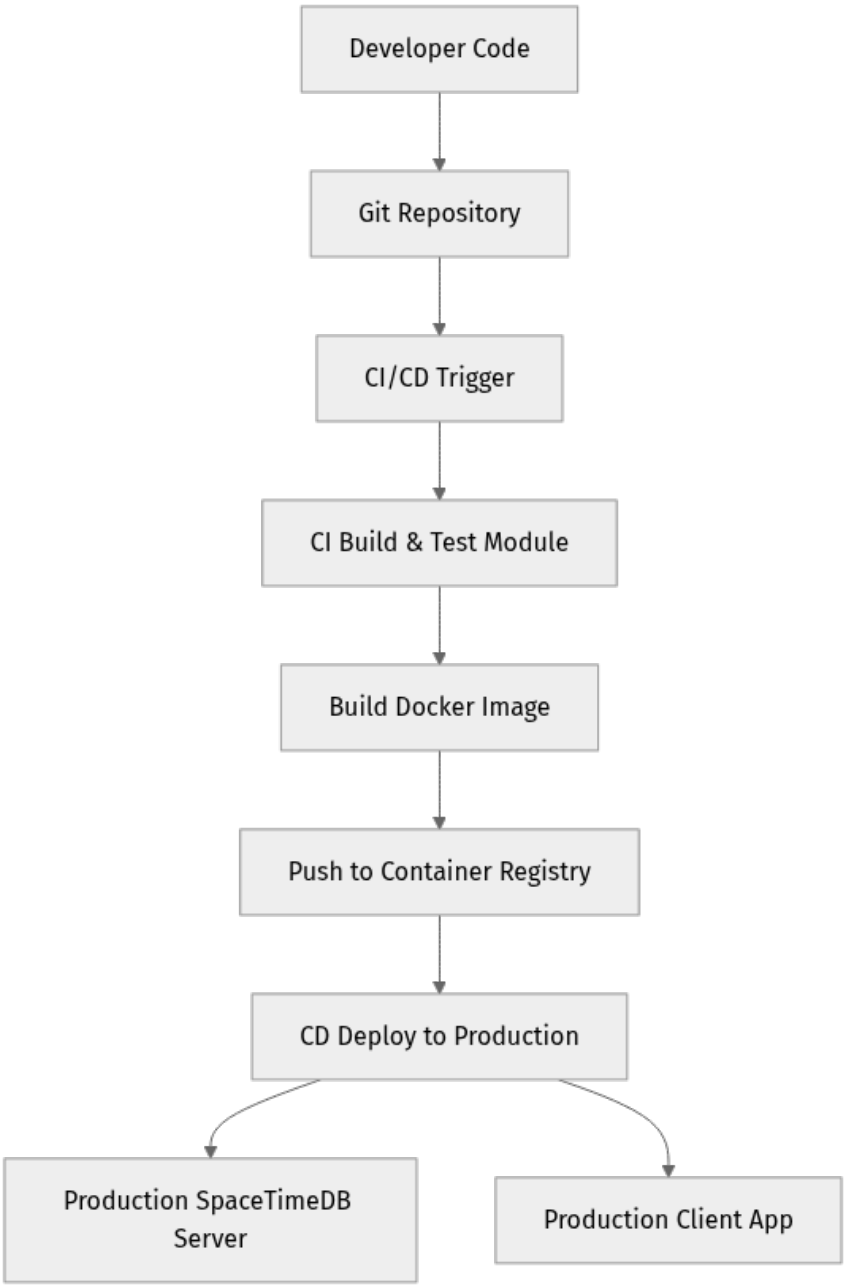


Figure 17.1: Simplified CI/CD Pipeline for SpaceTimeDB applications

3. Observability: Seeing What's Happening in Production

Once your application is live, you need to know if it's healthy, performing well, and if users are encountering issues. This is where observability comes in, typically through logging, monitoring, and alerting.

A. Logging

What is it? Recording events, errors, and information about your application's execution. **Why is it important?** Essential for debugging issues, understanding application behavior, and auditing. **How it functions:**

- **Structured Logging:** Instead of plain text, log messages are formatted (e.g., JSON) with key-value pairs (`{"level": "info", "message": "User connected", "user_id": "abc"}`). This makes logs easier to search and analyze with log management tools (e.g., ELK Stack, Splunk, Datadog).
- **Log Levels:** Use different severities (DEBUG, INFO, WARN, ERROR, CRITICAL) to control the verbosity. In production, you might only log INFO level and above to reduce noise.

SpaceTimeDB reducers and client logic should use structured logging. Rust's `log` crate combined with a formatter like `env_logger` or `tracing` can achieve this.

B. Monitoring

What is it? Collecting and analyzing metrics (numerical data points) about your application and infrastructure. **Why is it important?** Provides insights into performance, resource utilization, and overall system health. Helps identify trends and potential problems before they become critical. **How it functions:**

- **Infrastructure Metrics:** CPU usage, memory consumption, network I/O, disk space for your SpaceTimeDB server and client hosts.
- **Application Metrics:** SpaceTimeDB-specific metrics like:
  - Number of active client connections.
  - Reducer execution times.
  - Number of reducer invocations.
  - Database query latency.
  - Error rates from reducers or client synchronization.
- **Tools:** Prometheus, Grafana, Datadog, New Relic.

C. Alerting

What is it? Automatically notifying relevant personnel when specific metrics or log patterns indicate a problem. **Why is it important?** Allows for proactive response to issues, minimizing downtime and impact on users. **How it functions:** You define thresholds for your monitored metrics (e.g., "CPU usage > 80% for 5 minutes," "Error rate > 5%"). When a threshold is breached, an alert is triggered via email, Slack, PagerDuty, etc.

4. Security in Production: Protecting Your Application and Data

Security is paramount in production. A single vulnerability can compromise your entire system.

A. Network Security

What is it? Protecting your application's network access. **Why is it important?** Prevents unauthorized access and attacks. **How it functions:**

- **Firewalls:** Restrict incoming and outgoing network traffic to only what's necessary.
- **Virtual Private Clouds (VPCs):** Isolate your cloud resources in a private network.
- **TLS/SSL:** Encrypt all data in transit (e.g., between clients and the SpaceTimeDB server, between your SpaceTimeDB server and other backend services). SpaceTimeDB clients typically connect over WebSockets, which should always be secured with WSS (WebSocket Secure).

B. Authentication and Authorization

What is it?

- **Authentication:** Verifying a user's identity ("Who are you?").
- **Authorization:** Determining what an authenticated user is allowed to do ("What can you access?"). **Why is it important?** Controls access to your SpaceTimeDB data and reducers, preventing unauthorized operations. **How it functions:**
- **External Identity Providers (IdPs):** Integrate with services like Auth0, AWS Cognito, Google Firebase Auth, or OAuth2/OpenID Connect providers. Your client application authenticates users with the IdP, receives a token, and then uses this token to connect to SpaceTimeDB.
- **SpaceTimeDB Permissions:** Use SpaceTimeDB's built-in permission system (if available, check current docs) or implement custom authorization logic within your reducers to check user roles/permissions based on data stored in tables.

C. Data at Rest Encryption

What is it? Encrypting data when it's stored on disk. **Why is it important?** Protects sensitive data even if the underlying storage is compromised. **How it functions:** Cloud providers typically offer disk encryption by default for their storage services. Ensure your SpaceTimeDB server's data directories are on encrypted volumes.

D. Secrets Management

What is it? Securely storing and accessing sensitive information like API keys, database credentials, and private keys. **Why is it important?** Prevents hardcoding secrets in your code, which is a major security risk. **How it functions:** Use dedicated secrets management services (e.g., AWS Secrets Manager, HashiCorp Vault, Kubernetes Secrets) that encrypt and control access to your secrets. Your application retrieves secrets from these services at runtime.

5. High Availability & Disaster Recovery

Ensuring your application remains available even during failures and can recover from catastrophic events.

A. High Availability (HA)

What is it? Designing your system to operate continuously without interruption for a long time. **Why is it important?** Minimizes downtime, crucial for real-time applications where users expect constant connectivity. **How it functions:**

- **Redundancy:** Running multiple instances of your SpaceTimeDB server (if SpaceTimeDB supports clustering/replication for HA, which is common for real-time databases).
- **Load Balancing:** Distributing client connections across multiple SpaceTimeDB instances.
- **Failover:** Automatically switching to a healthy instance if one fails.

B. Disaster Recovery (DR)

What is it? A plan to recover your application and data after a major outage or disaster (e.g., entire data center failure). **Why is it important?** Ensures business continuity and prevents data loss. **How it functions:**

- **Backups:** Regularly backing up your SpaceTimeDB database (both schema and data) to an offsite location.
- **Recovery Point Objective (RPO):** The maximum amount of data you are willing to lose (e.g., 5 minutes of data).
- **Recovery Time Objective (RTO):** The maximum amount of time you can tolerate for your application to be down.
- **Testing:** Regularly testing your backup and restore procedures to ensure they work.

6. Schema Evolution & Migrations

Your application will evolve, and so will your database schema. How do you handle changes without disrupting live users?

What is it? The process of applying changes to your SpaceTimeDB schema (tables, reducers, views) in a controlled and safe manner in production. **Why is it important?** Allows your application to evolve while maintaining data integrity and minimizing downtime. **How it functions:**

- **Backward Compatibility:** Design schema changes so older client versions can still interact with the new schema (e.g., adding nullable columns, avoiding renaming columns).
- **Migration Scripts:** Use version-controlled scripts to apply schema changes. For SpaceTimeDB, this might involve using the CLI to apply new module versions that define schema changes.
- **Phased Rollouts:** Gradually deploy new versions of your application and schema, monitoring for issues.
- **Rollback Plan:** Always have a plan to revert to the previous state if a migration causes problems.

7. Performance Optimization in Production

While you optimize during development, production reveals real-world bottlenecks.

What is it? Continuously identifying and addressing performance issues in your live application. **Why is it important?** Ensures a smooth, responsive user experience and efficient resource utilization. **How it functions:**

- **Indexing:** Ensure appropriate indexes are defined on your SpaceTimeDB tables to speed up common queries.
- **Efficient Reducers:** Profile your reducers to identify slow operations. Optimize data access patterns.
- **Client-Side Optimization:** Minimize data fetching, optimize client-side rendering, and manage subscription lifecycles effectively.
- **Load Testing:** Simulate high user traffic to identify breaking points before they occur in production.

Step-by-Step Implementation: Dockerizing Your SpaceTimeDB Module and Basic Logging

Let's put some of these concepts into practice. We'll take a simple SpaceTimeDB module and containerize it using Docker, then demonstrate how to use environment variables for configuration and implement basic structured logging.

For this exercise, we'll assume you have a basic SpaceTimeDB module (e.g., from a previous chapter) that defines a simple table and a reducer.

First, let's create a minimal `lib.rs` for our SpaceTimeDB module that logs a message when a reducer is called.

1. Prepare Your SpaceTimeDB Module

Create a new directory for your module, let's call it `my_stdb_module`. Inside `my_stdb_module`, create a `Cargo.toml` and `src/lib.rs`.

`my_stdb_module/Cargo.toml`:

```
[package]
name = "my_stdb_module"
version = "0.1.0"
edition = "2021"

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

[dependencies]
# Use the latest stable version of SpacetimeDB as of 2026-03-14
# Check official SpacetimeDB documentation for the most accurate current
version
spacetimedb = "2.2.0" # Placeholder, verify latest on spacetimedb.com/docs or
GitHub releases
log = "0.4.21" # For logging
env_logger = "0.11.3" # For basic environment-based logger
```

Note: Please verify the actual latest stable versions for *`spacetimedb`*, *`log`*, and *`env_logger`* on their respective official documentation or crates.io as of your build date.

`my_stdb_module/src/lib.rs`:

```rust
use spacetimedb::{
    spacetimedb,
    table,
    ReducerContext,
    log,
};

// Initialize logger once
#[spacetimedb::init]
fn init() {
    // Only initialize env_logger if it hasn't been initialized yet
    // This prevents "logger already initialized" errors in certain
environments
    let _ = env_logger::builder()
        .filter_level(log::LevelFilter::Info) // Default to INFO level
        .is_test(true) // Disable color output when running as a test
        .try_init();

    log::info!("SpaceTimeDB module initialized!");
}


// Define a simple table
#[table]
pub struct Counter {
    #[primarykey]
    pub id: u32,
    pub value: u32,
}

// Define a reducer that increments the counter and logs
#[spacetimedb(reducer)]
pub fn increment_counter(ctx: ReducerContext, id: u32) {
    // Log the reducer call with structured data
    log::info!(
        target: "my_stdb_module::reducer",
        "Reducer `increment_counter` called by user: {:?} with id: {}",
        ctx.sender,
        id
    );

    let mut counter = Counter::filter_by_id(&id).unwrap_or_else(|| Counter {
id, value: 0 });
    counter.value += 1;
    counter.insert();

    log::info!("Counter with id {} incremented to {}", id, counter.value);
}
```

Explanation: 1. We include `log` and `env_logger` crates. 2. The `init` function ensures `env_logger` is set up when the module starts. We use `try_init()` to avoid panicking if the logger is already initialized (e.g., in a test environment). We set a default `Info` level. 3. The `increment_counter` reducer now uses `log::info!` to output messages. We're using `target` for better log categorization and including relevant data like `ctx.sender` and `id` in the message.

2. Compile Your SpaceTimeDB Module

Navigate to `my_stdb_module` in your terminal and compile it:

```
spacetime-cli build
```

This will create a `.wasm` file in your `target/spacetime` directory. This is the compiled SpaceTimeDB module that the SpaceTimeDB server will load.

3. Dockerize the SpaceTimeDB Server with Your Module

Now, let's create a `Dockerfile` to run the SpaceTimeDB server and load our module. We'll use a multi-stage build to keep the final image small.

Create a file named `Dockerfile` in the root of your project (one level up from `my_stdb_module`):

```
# Stage 1: Build the SpaceTimeDB module
FROM rust:1.76.0-slim-bookworm AS module-builder

WORKDIR /app
COPY my_stdb_module ./my_stdb_module
RUN cd my_stdb_module && cargo build --target wasm32-unknown-unknown --release

# Stage 2: Create the final SpaceTimeDB server image
# Use the official SpacetimeDB CLI image or a minimal base image
# As of 2026-03-14, SpacetimeDB v2.x is stable.
# Assume an official CLI image exists, or build from source if not.
# For simplicity, we'll use a common Linux base and install the CLI if needed.
# In a real scenario, you'd use clockworklabs/spacetimedb-cli or similar if
available.
FROM debian:bookworm-slim

# Install SpaceTimeDB CLI (replace with official installation method if
different)
# This is a placeholder; consult official SpacetimeDB docs for production CLI
installation.
# For this example, we'll assume the CLI is installed and available.
# A more robust Dockerfile would download and install the specific CLI version.
# For educational purposes, let's simulate having `spacetime-cli` installed.
# In a true production setup, you'd likely have a pre-built SpaceTimeDB server
binary
# or a Docker image provided by Clockwork Labs.
# For this example, we'll focus on running the server and loading *your*
module.
# Let's assume we copy a pre-installed `spacetime-cli` or `spacetimedb-server`
binary.
# For this exercise, we'll use the `spacetime-cli` to run the `spacetime-db`
command.

# Install necessary runtime dependencies for the CLI if it's a Rust binary
# (e.g., openssl if linked dynamically)
# RUN apt-get update && apt-get install -y --no-install-recommends \
#     libssl-dev \
#     && rm -rf /var/lib/apt/lists/*

# For demonstration, let's assume `spacetime-cli` is already in a base image or
pre-installed.
# If not, you'd download and install it here.
# For this example, we'll simulate it by copying a local binary (not ideal for
real prod, but works for concept)
# Or, assume `clockworklabs/spacetimedb` is the base image. Let's make this
explicit.

# Let's use a more realistic approach, assuming SpacetimeDB provides a base
image.
# If no official base image, we'd manually install the `spacetime-cli` binary.
# For now, let's simplify and assume `spacetime-cli` is present.
# In a real scenario, you'd use: FROM clockworklabs/spacetimedb:2.x.y

# For this guide, let's assume we're running `spacetime-cli db` within a
generic Linux container.
# This would require `spacetime-cli` to be available.
# A more direct approach is to install `spacetime-cli` here:
RUN apt-get update && apt-get install -y curl && \
    curl -L https://install.spacetimedb.com | sh && \
    apt-get remove -y curl && apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*
```

```
ENV PATH="/root/.spacetime/bin:${PATH}"

WORKDIR /app

# Copy the compiled module from the builder stage
COPY --from=module-builder /app/my_stdb_module/target/wasm32-unknown-unknown/
release/my_stdb_module.wasm ./modules/my_stdb_module.wasm

# Expose the default SpacetimeDB port
EXPOSE 3000

# Command to run the SpaceTimeDB server
# We'll load our module and use environment variables for persistence
ENTRYPOINT ["spacetime-cli", "db", "--module", "./modules/my_stdb_module.wasm"]
# Use `--data-dir` for persistence. This should be a volume in production.
# We'll set this via an environment variable in docker-compose.
```

Explanation: 1. **Multi-stage build:**

- **`module-builder` stage:** Uses a Rust image to compile our `my_stdb_module` into a `.wasm` file. This keeps the final image lean.
- **Final stage:** Uses a minimal `debian:bookworm-slim` base. 2. **SpaceTimeDB CLI Installation:** We use `curl | sh` for `spacetime-cli` installation, which is a common pattern for CLI tools, but in production, you might prefer downloading a specific binary version for better control and security. 3. **Copy Module:** The compiled `.wasm` module is copied from the `module-builder` stage into the final image. 4. **Expose Port:** SpaceTimeDB typically listens on port 3000. 5. **`ENTRYPOINT`:** This defines the command that runs when the container starts. We're telling `spacetime-cli db` to load our compiled module. We'll manage persistence via Docker volumes and environment variables.

4. Create a `docker-compose.yml` for Local Testing

To easily run and manage our SpaceTimeDB server, we'll use `docker-compose`. This allows us to define services, networks, and volumes.

Create a `docker-compose.yml` file in the same directory as your `Dockerfile`:

```
version: '3.8'

services:
  spacetimedb:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    environment:
      # Production-like settings
      RUST_LOG: "info,my_stdb_module=debug" # Log INFO by default, but DEBUG
for our module
      SPACETIMEDB_DATA_DIR: "/data" # Where SpaceTimeDB will store its
persistent data
    volumes:
      - stdb_data:/data # Mount a named volume for persistent data

volumes:
  stdb_data:
```

Explanation: 1. **`spacetimedb` service:** Defines our SpaceTimeDB server. 2. **`build`:** Tells Docker Compose to build the image using our `Dockerfile`. 3. **`ports`:** Maps container port 3000 to host port 3000, allowing client connections. 4. **`environment`:** * `RUST_LOG`: This is crucial for controlling logging. We set it to `info` for general output but `debug` specifically for our `my_stdb_module` target. In a true production setup, you might remove `debug` for your module unless actively troubleshooting. * `SPACETIMEDB_DATA_DIR`: We define a directory inside the container for SpaceTimeDB's data. This allows us to mount a volume for persistence. 5. **`volumes`:** We mount a named Docker volume (`stdb_data`) to the `/data` directory inside the container. This ensures that even if the container is removed, your SpaceTimeDB data persists. This is critical for production!

5. Run Your Dockerized SpaceTimeDB Server

Now, from your project root (where `Dockerfile` and `docker-compose.yml` are), run:

```
docker compose up --build
```

You should see output indicating the SpaceTimeDB server starting, and then your `init` function's log message:

```
spacetimedb-1  | [2026-03-14T10:00:00Z INFO  my_stdb_module] SpaceTimeDB module
initialized!
```

6. Test with a Client (or CLI)

Open another terminal and use the `spacetime-cli` to connect and invoke your reducer.

```
spacetime-cli connect ws://localhost:3000 # Connect to your local SpaceTimeDB instance
spacetime-cli call increment_counter 1 # Call the reducer with ID 1
```

Back in your Docker logs, you should now see the `INFO` and `DEBUG` level messages from your reducer:

```
spacetimedb-1  | [2026-03-14T10:00:05Z INFO  my_stdb_module::reducer] Reducer `increment_counter` called by user: Some(0x...) with id: 1
spacetimedb-1  | [2026-03-14T10:00:05Z INFO  my_stdb_module] Counter with id 1 incremented to 1
```

Notice how `RUST_LOG` makes our specific module's logs more verbose, while keeping the general SpaceTimeDB server logs at `INFO` level. This is a powerful technique for debugging in production without overwhelming your log systems.

Mini-Challenge: Adding a Health Check Endpoint

A crucial part of production readiness is being able to tell if your application is actually running and healthy. Load balancers and orchestration systems (like Kubernetes) use health checks to determine if they should send traffic to an instance.

Your Challenge: Modify your `my_stdb_module` to include a simple "health check" mechanism. While SpaceTimeDB itself doesn't expose HTTP endpoints directly from modules, you can create a simple `health_check` reducer that just returns `true`. A client (or an automated script) could then call this reducer to confirm the module is loaded and operational.

Steps: 1. Add a new reducer called `health_check` to `my_stdb_module/src/lib.rs`. 2. This reducer should take no arguments and simply log an `INFO` message indicating it was called, then return `true`. 3. Recompile your module (`spacetime-cli build`). 4. Rebuild and restart your Docker container (`docker compose up --build`). 5. Use `spacetime-cli call health_check` to verify it works and observe the logs.

Hint: Remember that reducers can return values. The `ReducerContext` is optional if not used.

```
// Example of a health_check reducer
#[spacetimedb(reducer)]
pub fn health_check() -> bool {
    log::info!(target: "my_stdb_module::health_check", "Health check reducer
called.");
    true
}
```

What to Observe/Learn: * How to extend your SpaceTimeDB module with simple diagnostic reducers. * How logs provide visibility into the module's internal state and activity. * The concept of a health check, even if simplified for SpaceTimeDB's reducer-based model.

Common Pitfalls & Troubleshooting

1. **Misconfigured Environment Variables:**

- **Pitfall:** Your application behaves differently in production than in development, or fails to start, due to incorrect environment variable values (e.g., wrong database URL, missing API key).
- **Troubleshooting:**
- **Verify:** Double-check the environment variables set in your deployment configuration (Docker Compose, Kubernetes deployment, cloud service settings).
- **Log:** Temporarily increase logging verbosity to `DEBUG` and log the values of critical environment variables at startup (be careful not to log sensitive secrets!).
- **Access:** Ensure your application has the necessary permissions to read environment variables in the deployed environment.

1. **Insufficient Logging/Monitoring:**

- **Pitfall:** Your application crashes or performs poorly in production, but you have no idea why because there aren't enough logs or metrics.
- **Troubleshooting:**
- **Structured Logs:** Implement structured logging from the start. It's much harder to add later.
- **Log Levels:** Use appropriate log levels (`INFO`, `WARN`, `ERROR`) for production. Enable `DEBUG` only when actively troubleshooting.

- **Key Metrics:** Identify critical metrics (CPU, memory, network, SpaceTimeDB connections, reducer latency, error rates) and set up monitoring and alerting for them. Don't wait for an incident to realize you're blind.

1. **Schema Migration Issues:**

- **Pitfall:** Deploying a new SpaceTimeDB module with schema changes causes data loss, application errors, or downtime for existing users.
- **Troubleshooting:**
- **Test Migrations:** Always test your schema migration scripts thoroughly in a staging environment that mirrors production data.
- **Backward Compatibility:** Design schema changes to be backward-compatible whenever possible. If not, plan for a clear deprecation and transition strategy.
- **Atomic Deployments:** Use deployment strategies that apply schema changes and update application code in an atomic way or with zero downtime (e.g., blue/green deployments).
- **Rollback Plan:** Have a clear rollback strategy in case a migration fails. This might involve reverting to a previous module version and restoring a database backup.

Summary

Phew! You've navigated the complexities of taking your SpaceTimeDB application to production. This is where the rubber meets the road, and understanding these best practices is key to success.

Here are the key takeaways from this chapter:

- **Environment Separation:** Always distinguish between development and production configurations, primarily using environment variables for sensitive data and dynamic settings.
- **Containerization is King:** Dockerize your SpaceTimeDB modules and client applications for consistent, portable, and isolated deployments.
- **Automate with CI/CD:** Implement Continuous Integration and Deployment pipelines to streamline your release process, ensuring faster, more reliable deployments.

- **Observability is Non-Negotiable:** Leverage structured logging, comprehensive monitoring, and proactive alerting to understand your application's health and performance in real-time.
- **Security First:** Implement robust network security, integrate proper authentication and authorization, encrypt data, and manage secrets securely.
- **Plan for the Worst:** Design for High Availability to minimize downtime and develop a Disaster Recovery plan, including regular backups and tested restore procedures.
- **Manage Evolution:** Handle schema changes gracefully with backward compatibility and well-tested migration strategies.
- **Optimize Continuously:** Keep an eye on performance in production, using indexes, efficient reducer design, and client-side optimizations.

You now have a solid foundation for deploying and maintaining SpaceTimeDB applications in a production environment. This knowledge empowers you to build not just functional, but also resilient and scalable real-time systems.

What's next? With a fully deployed and observable SpaceTimeDB application, you're ready to explore even more advanced topics, such as deep dives into specific cloud provider integrations, advanced scaling patterns, or perhaps even contributing back to the SpaceTimeDB ecosystem!

References

1. **SpaceTimeDB Official Documentation:** https://spacetimedb.com/docs
2. **Rust `log` crate:** https://docs.rs/log/latest/log/
3. **Rust `env_logger` crate:** https://docs.rs/env_logger/latest/env_logger/
4. **Docker Documentation:** https://docs.docker.com/
5. **Docker Compose Documentation:** https://docs.docker.com/compose/
6. **Continuous Integration and Delivery (CI/CD) Concepts:** https://www.redhat.com/en/topics/devops/what-is-ci-cd
7. **The Twelve-Factor App (Environment Variables):** https://12factor.net/config

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 10

Chapter 2: Your First SpaceTimeDB Project: Setup and Workflow

Chapter 2: Your First SpaceTimeDB Project: Setup and Workflow

Welcome back, aspiring real-time architect! In Chapter 1, we explored the "why" behind SpaceTimeDB, understanding its unique approach to unifying database, backend logic, and real-time synchronization. Now, it's time to roll up our sleeves and dive into the "how."

This chapter is your hands-on initiation into the SpaceTimeDB universe. We'll guide you through setting up your development environment, creating your very first SpaceTimeDB project, defining a simple database schema, and writing server-side logic that modifies your data. By the end, you'll have a running SpaceTimeDB instance on your local machine, ready to power real-time applications. Get ready to build, learn, and have some fun!

The SpaceTimeDB CLI: Your Command Center

At the heart of SpaceTimeDB development is the Command Line Interface (CLI). Think of it as your primary tool for everything from creating new projects and compiling your backend logic to running your local SpaceTimeDB instance and interacting with your database. The CLI streamlines the entire development workflow, making it easy to manage your SpaceTimeDB applications.

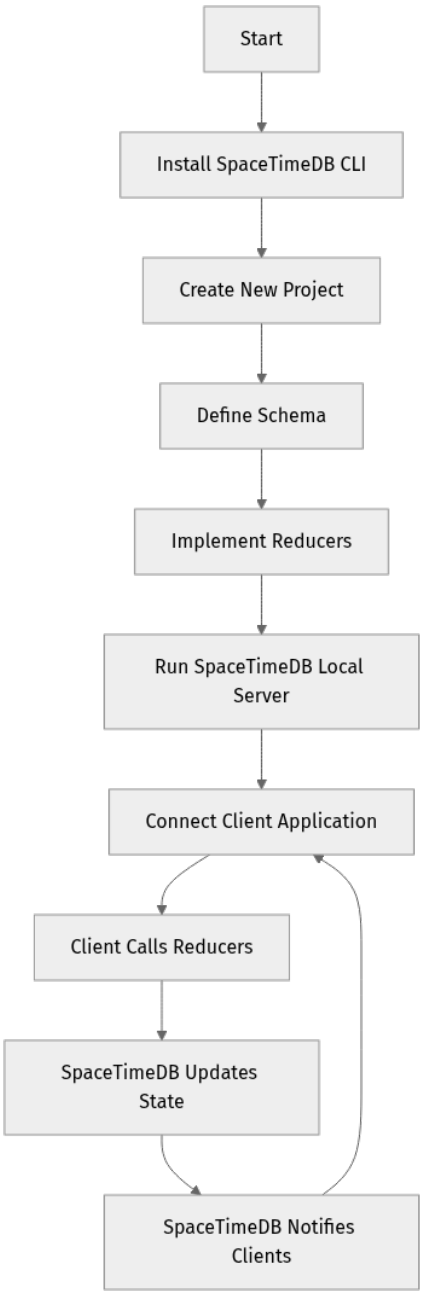
Understanding the SpaceTimeDB Development Workflow

Before we start typing commands, let's get a mental picture of the typical SpaceTimeDB development cycle. It's a bit different from traditional backend development, but incredibly efficient once you get the hang of it:

1. **Initialize Project:** Use the CLI to scaffold a new SpaceTimeDB project. This sets up the basic directory structure and configuration files.
2. **Define Schema:** Describe your application's data structure using Rust types. These types become your database tables.

3. **Implement Reducers:** Write Rust functions (called "reducers") that define how your application's state can change. These are your server-side "API endpoints" or "game actions."
4. **Run Local Instance:** Start a local SpaceTimeDB server using the CLI. It compiles your Rust code, creates the database, and watches for changes.
5. **Connect Clients:** Develop frontend applications (web, mobile, game engines) that connect to your SpaceTimeDB instance and call your reducers to modify state, and subscribe to tables to receive real-time updates.

Here's a simplified visual representation of this workflow:



Step-by-Step Implementation: Building Your First Project

Let's get our hands dirty!

1. Prerequisites: What You'll Need

SpaceTimeDB leverages Rust for its server-side logic. So, the first thing you'll need is a working Rust environment.

- **Rust and Cargo:** If you don't have Rust installed, the easiest way is through `rustup`. Open your terminal and run: `bash curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh` Follow the on-screen instructions. This will install `rustc` (the Rust compiler) and `cargo` (Rust's package manager and build tool).
- **Version Check:** After installation, verify with: `bash rustc --version cargo --version` As of 2026-03-14, you should ideally be on a stable Rust version like **Rust 1.77.x** or newer.

2. Installing the SpaceTimeDB CLI

Now, let's install the core tool: the SpaceTimeDB CLI. This is a single binary that simplifies your development.

- **Installation Command:** Open your terminal and run: `bash cargo install spacetimedb-cli` This command uses Cargo to fetch and compile the `spacetimedb-cli` package from crates.io and installs it to your Cargo bin directory (usually `~/.cargo/bin`). Make sure this directory is in your system's `PATH`.
- **Verifying Installation:** Once installed, confirm it's working by checking its version: `bash spacetime --version` You should see output similar to `spacetime-cli v2.2.0` (or a later `v2.x` version as of 2026-03-14). This confirms the CLI is correctly installed and accessible.

3. Creating Your First SpaceTimeDB Project

With the CLI ready, let's scaffold a new project. We'll create a simple "To-Do List" backend.

1. **Choose a Directory:** Navigate to a directory where you want to create your project. `bash cd ~/projects`
2. **Create the Project:** Use the `spacetime new` command: `bash spacetime new my_todo_app` The CLI will create a new directory named `my_todo_app` and populate it with initial files.

3. **Explore the Project Structure:** Navigate into your new project: `bash cd my_todo_app` Take a look around. You'll see something like this: `my_todo_app/ ├── src/ │ └── lib.rs └── Spacetime.toml`
    - `src/lib.rs`: This is where your Rust code for defining tables and reducers will live. It's the "brain" of your SpaceTimeDB backend.
    - `Spacetime.toml`: This is the SpaceTimeDB configuration file. It tells the CLI how to build and run your project.

4. Understanding `Spacetime.toml`

Open `Spacetime.toml` in your favorite code editor. It's a TOML file that configures your SpaceTimeDB instance.

```
# Spacetime.toml
[spacetime]
module_name = "my_todo_app"
# The full path to the Rust module that defines your tables and reducers.
# This usually points to your `src/lib.rs` file.
module_path = "src/lib.rs"

# [client_api]
# This section is for configuring client API generation,
# which we'll cover in a later chapter.
# For now, we'll leave it commented out or empty.
```

- `module_name`: This is the name of your SpaceTimeDB module. It's typically the same as your project directory name.
- `module_path`: This specifies the path to your main Rust source file (`lib.rs`) where your database schema and logic are defined.

For now, these defaults are perfect. We'll revisit this file in future chapters for advanced configurations like client API generation.

5. Defining Your First Schema: The `Todo` Table

Now, let's define our first database table. In SpaceTimeDB, you define your tables using Rust structs.

Open `src/lib.rs` and replace its contents with the following:

```
// src/lib.rs
use spacetimedb::{spacetimedb, Identity, ReducerContext, SpacetimeType, Spaceti
meTable};

// 1. Define your table as a Rust struct.
//    The `#[derive(...)]` macros are crucial for SpaceTimeDB to recognize this
as a table.
#[spacetimedb(table)]
#[derive(SpacetimeType, SpacetimeTable)]
pub struct Todo {
    // 2. Define fields for your table.
    //    `#[primarykey]` designates this field as the unique identifier for
each row.
    #[primarykey]
    pub id: u32,
    pub description: String,
    pub completed: bool,
    pub created_by: Identity, // Stores the identity of the user who created
the todo.
}

// We'll add our reducer here next!
```

Let's break down what we just added:

- `use spacetimedb::{...}`: This line imports necessary traits and macros from the `spacetimedb` crate.
- `#[spacetimedb(table)]`: This attribute macro tells SpaceTimeDB that the `Todo` struct should be treated as a database table. It's essential!
- `#[derive(SpacetimeType, SpacetimeTable)]`: These derive macros implement traits required by SpaceTimeDB for serialization/deserialization and table management.
    - `SpacetimeType`: Allows the struct to be used as a type within SpaceTimeDB's system.
    - `SpacetimeTable`: Provides the necessary methods for the struct to function as a table.
- `pub struct Todo { ... }`: This defines our `Todo` table.
- `#[primarykey]`: This attribute marks the `id` field as the primary key. Every table must have exactly one primary key. It ensures each `Todo` item has a unique `id`.
- `pub id: u32`: An unsigned 32-bit integer for the todo's ID.
- `pub description: String`: A `String` to hold the task description.
- `pub completed: bool`: A boolean to track if the task is done.

- `pub created_by: Identity`: This is a special SpaceTimeDB type that represents the unique identifier of a connected client. This is incredibly powerful for tracking who performed an action or owns data!

Why Rust for Schema? By defining your schema directly in Rust, you get strong type safety from the very beginning. Your database schema and your backend logic are defined in the same language, in the same project, leading to fewer mismatches and a more robust system.

6. Implementing Your First Reducer: `create_todo`

Now that we have a `Todo` table, how do we add data to it? This is where **reducers** come in. Reducers are functions defined in your `src/lib.rs` that encapsulate all the logic for modifying your SpaceTimeDB's state. When a client wants to change something, they "call" a reducer.

Add the following code below your `Todo` struct in `src/lib.rs`:

```
// src/lib.rs (continued)
// ... (your Todo struct code here) ...

// 3. Define a reducer function.
//    `#[spacetimedb(reducer)]` marks this function as a reducer callable by
clients.
#[spacetimedb(reducer)]
pub fn create_todo(ctx: ReducerContext, id: u32, description: String) ->
Result<(), String> {
    // 4. Access the Identity of the client calling this reducer.
    let creator_identity = ctx.identity();

    // 5. Check if a todo with this ID already exists.
    if Todo::filter_by_id(&id).count() > 0 {
        return Err(format!("Todo with ID {} already exists.", id));
    }

    // 6. Insert a new Todo item into the database.
    Todo::insert(Todo {
        id,
        description,
        completed: false,
        created_by: creator_identity,
    });

    // 7. Return Ok(()) for success, or an Err(String) for failure.
    Ok(())
}
```

Let's unpack this reducer:

- `#[spacetimedb(reducer)]`: This attribute macro is crucial. It tells SpaceTimeDB that `create_todo` is a function that clients can invoke to change the database state.

- `pub fn create_todo(ctx: ReducerContext, id: u32, description: String) -> Result<(), String>`:
  - `ctx: ReducerContext`: Every reducer receives a `ReducerContext`. This provides access to important information about the current call, such as the `Identity` of the client making the request.
  - `id: u32, description: String`: These are the arguments that a client will pass when calling this reducer. They become the data for our new `Todo`.
  - `-> Result<(), String>`: Reducers typically return a `Result`. `Ok(())` indicates success, and `Err(String)` indicates an error with a message.
- `let creator_identity = ctx.identity();`: We extract the `Identity` of the client who called this reducer. This is a powerful feature for security and data ownership.
- `if Todo::filter_by_id(&id).count() > 0 { ... }`: Before inserting, we check if a todo with the given `id` already exists. `Todo::filter_by_id()` is a generated helper function that allows querying the `Todo` table by its primary key.
- `Todo::insert(Todo { ... });`: This is how you add a new row to the `Todo` table. You create an instance of your `Todo` struct and call the static `insert` method on the `Todo` type.
- `Ok(())`: If everything goes well, we return `Ok(())` to signal success.

Why Reducers? Reducers centralize your application's logic. All state changes must go through a reducer. This ensures:

- **Determinism:** Given the same initial state and the same sequence of reducer calls, the final state will always be the same. This is key for SpaceTimeDB's consistency model.
- **Validation:** You can put all your input validation and business logic directly in your reducers.
- **Security:** By controlling what actions are allowed (via reducers), you enforce your application's rules on the server.

7. Running Your SpaceTimeDB Local Server

With our schema and reducer defined, it's time to bring our SpaceTimeDB instance to life!

1. **Run the Development Server:** In your `my_todo_app` project directory, open your terminal and run: `bash spacetime dev` The `spacetime dev` command does several things:
    - It compiles your Rust code (`src/lib.rs`).
    - It starts a local SpaceTimeDB server (typically listening on `ws://localhost:3000`).
    - It watches your `src/lib.rs` file for changes. If you save changes, it will automatically recompile and restart the server, providing a fast development loop.

    You should see output indicating compilation success and the server starting: `Compiling my_todo_app v0.1.0 (~/projects/my_todo_app) Finished dev [unoptimized + debuginfo] target(s) in X.XXs Running `target/debug/my_todo_app` [INFO] SpacetimeDB server started on ws://localhost:3000 [INFO] Watching for changes in src/lib.rs...` Keep this terminal window open; your SpaceTimeDB server is now running!

8. Interacting with the SpaceTimeDB Console

While your server is running, open another terminal window, navigate back to your `my_todo_app` directory, and let's use the `spacetime console` to interact with it.

```
spacetime console
```

This will open an interactive console that connects to your local SpaceTimeDB instance. You'll see a prompt like `spacetime-console>`.

- **Calling the `create_todo` reducer:** Let's add our first todo item! `spacetime-console> call create_todo 1 "Learn SpaceTimeDB"` You should see output indicating success: `Reducer 'create_todo' called successfully.` Let's add another one: `spacetime-console> call create_todo 2 "Build a real-time app"` - **Querying the `Todo` table:** Now, let's see if our todos are actually in the database. `spacetime-console> subscribe Todo` This command "subscribes" your console to the `Todo` table, meaning you'll get all current data and any future updates. You should immediately see the data you just inserted: `json [ { "id": 1,`

`"description": "Learn SpaceTimeDB", "completed": false, "created_by": "0x..." // Your client's identity }, { "id": 2, "description": "Build a real-time app", "completed": false, "created_by": "0x..." } ]` The `created_by` field will show a hexadecimal string, which is the unique `Identity` of your `spacetime console` session. Pretty neat, right?

You can exit the console by typing `exit` or pressing `Ctrl+D`.

Mini-Challenge: Extend Your To-Do App!

You've successfully set up your environment, defined a table, written a reducer, and interacted with your SpaceTimeDB instance. Now, let's solidify that knowledge with a small challenge.

Challenge: Modify your `Todo` table and `create_todo` reducer to include a new field: `due_date: Option<String>`.

1. **Add `due_date` to the `Todo` struct:** Make it an `Option<String>` so that a due date is optional.
2. **Update the `create_todo` reducer:** Modify its signature to accept `Option<String>` for the `due_date`. Update the `Todo::insert` call to include this new field.
3. **Test with `spacetime console`:** Restart your `spacetime dev` server (it should auto-restart if you save changes). Then, in `spacetime console`, try calling `create_todo` with and without a due date:
    - `call create_todo 3 "Buy groceries" "2026-03-15"`
    - `call create_todo 4 "Walk the dog" None`
    - Then `subscribe Todo` again to see your updated table.

Hint: Remember that `Option<String>` in Rust can be `Some("your_date_string".to_string())` or `None`.

What to Observe/Learn: * How easily you can evolve your schema. * How reducers adapt to schema changes. * How to handle optional fields in Rust and SpaceTimeDB.

Common Pitfalls & Troubleshooting

1. **`spacetime` command not found:**

- **Issue:** Your system's `PATH` environment variable might not include Cargo's bin directory (`~/.cargo/bin`).

- **Solution:** Ensure you've sourced your shell's profile after installing Rust (e.g., `source ~/.bashrc` or `source ~/.zshrc`). You might need to add `export PATH="$HOME/.cargo/bin:$PATH"` to your shell configuration file. 2. **Compilation Errors in `spacetime dev`:**
- **Issue:** Rust syntax errors, missing `use` statements, or incorrect `#[spacetimedb(...)]` attributes.
- **Solution:** Carefully read the error messages from the Rust compiler. They are usually very descriptive and point to the exact line and problem. Double-check your `src/lib.rs` against the provided code. 3. **Reducer not callable or incorrect arguments:**
- **Issue:** You might have typos in the reducer name when calling it from `spacetime console`, or you're providing the wrong number/type of arguments.
- **Solution:** Ensure the `call` command exactly matches the reducer's name and argument types. For `Option<String>`, remember to pass the string in quotes or `None`.

Summary

Congratulations! You've successfully completed your first SpaceTimeDB project. Here are the key takeaways from this chapter:

- **SpaceTimeDB CLI:** Your essential tool for project management, compilation, and running your local instance.
- **Project Structure:** A SpaceTimeDB project consists of `src/lib.rs` (your Rust logic) and `Spacetime.toml` (configuration).
- **Schema Definition:** You define your database tables using Rust structs annotated with `#[spacetimedb(table)]` and `#[primarykey]`. This brings strong type safety directly to your database.
- **Reducers:** These Rust functions, marked with `#[spacetimedb(reducer)]`, are the only way to modify your SpaceTimeDB's state. They provide a deterministic and controlled way to implement your application's logic.
- **Local Development:** The `spacetime dev` command compiles your code, starts a local server, and provides live reloading for a smooth development experience.
- **Console Interaction:** The `spacetime console` allows you to directly call reducers and subscribe to tables, making it easy to test and debug your backend.

You now have a solid foundation for building real-time applications with SpaceTimeDB. In the next chapter, we'll connect a simple web frontend to our `my_todo_app` and see how clients interact with our SpaceTimeDB instance to perform real-time data synchronization. Get ready to witness the magic!

References

1. **SpaceTimeDB Official Documentation:** https://spacetimedb.com/docs
2. **Rust Programming Language:** https://www.rust-lang.org/
3. **crates.io - spacetimedb-cli:** https://crates.io/crates/spacetimedb-cli
4. **SpaceTimeDB GitHub Repository:** https://github.com/clockworklabs/SpacetimeDB

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 11

Chapter 3: Structuring Your Data: Schema Design, Tables, and Relations

Introduction: The Blueprint for Your Real-time World

Welcome back, future SpaceTimeDB architects! In our previous chapters, we got acquainted with what SpaceTimeDB is and set up our development environment. Now, it's time to lay the foundation for your real-time applications: designing your database schema.

Just as an architect draws up blueprints before construction begins, you'll define your data's structure and relationships within SpaceTimeDB. This chapter is crucial because a well-designed schema isn't just about storing data; it's about enabling efficient real-time synchronization, consistent state management, and robust server-side logic. We'll explore how SpaceTimeDB combines the power of Rust with database table definitions to create a unified data model.

By the end of this chapter, you'll understand how to define tables, specify primary keys and indexes, and logically relate different pieces of data. You'll be ready to translate your application's data requirements into a SpaceTimeDB module, setting the stage for building dynamic, collaborative experiences.

Core Concepts: Defining Your Data Universe

In SpaceTimeDB, your database schema is much more than just a list of tables and columns. It's a Rust-based module that defines both your data's structure and, as we'll see in later chapters, your server-side logic (reducers). Let's break down the core components.

What is a SpaceTimeDB Schema?

Think of your SpaceTimeDB schema as the master plan for your entire application's backend. It's written in Rust, a powerful and safe programming language, and then compiled into WebAssembly (WASM). This WASM module is what SpaceTimeDB runs internally to manage your database, execute your logic, and ensure deterministic, consistent state across all connected clients.

Why Rust and WebAssembly?

- **Performance:** Rust compiles to highly optimized native code, offering incredible speed. WASM provides a near-native performance environment.
- **Safety & Determinism:** Rust's strong type system and ownership model prevent common programming errors, leading to more reliable and deterministic server-side logic. This determinism is key to SpaceTimeDB's ability to maintain a consistent shared state.
- **Unified Logic:** By defining both data and logic in the same Rust module, SpaceTimeDB creates a tightly integrated system where your database schema and business rules live side-by-side, simplifying development and deployment.

Tables: The Building Blocks of Your Data

At its heart, SpaceTimeDB organizes data into tables, much like traditional relational databases. Each table represents a collection of similar items, like "Users," "TodoItems," or "GameCharacters."

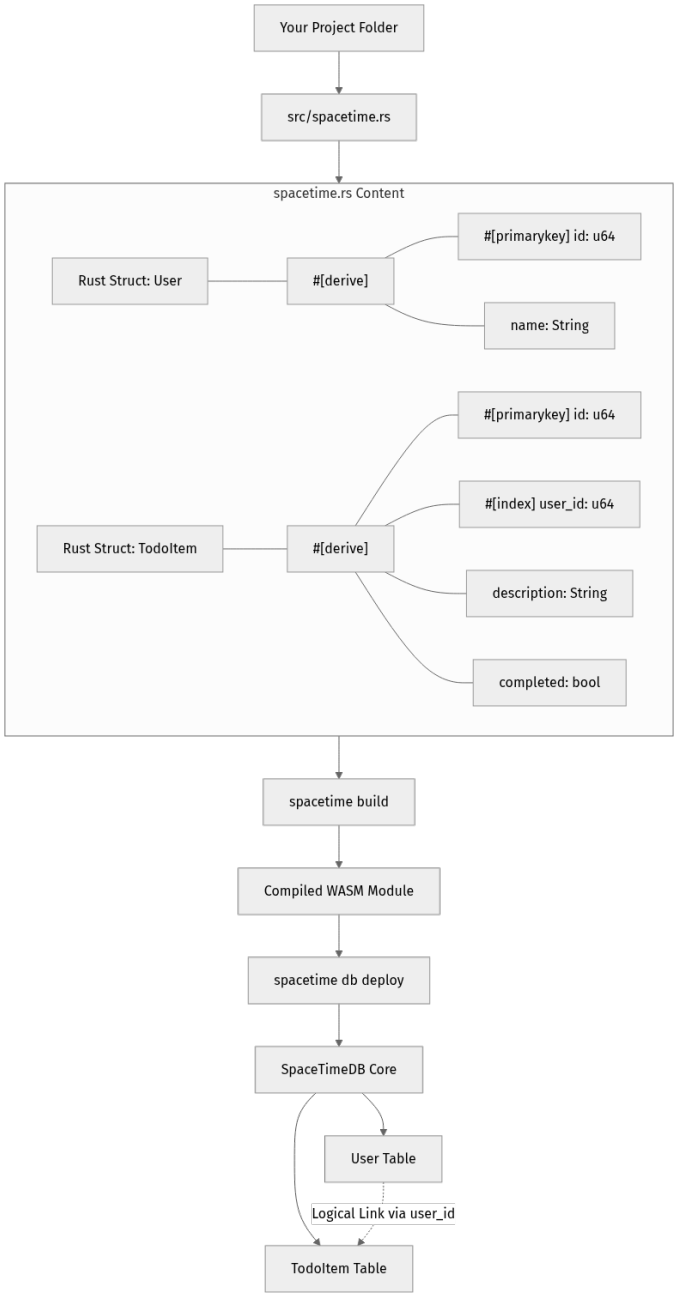
To define a table in SpaceTimeDB, you'll use a standard Rust `struct` and adorn it with special attributes (called "derive macros" in Rust) provided by the `SpacetimeDB` library.

Let's look at the key elements:

- **`#[derive(SpacetimeTable)]`**: This macro is essential. It tells the SpaceTimeDB compiler that your Rust `struct` should be treated as a database table. It automatically generates the necessary code to allow SpaceTimeDB to store, retrieve, and synchronize instances of this struct.
- **Fields and Data Types**: Inside your struct, you define fields using standard Rust data types (`u64`, `String`, `bool`, etc.) or even other custom structs (which can also be tables themselves or embedded data).
- **Primary Keys (`#[primarykey]`)**: Every table must have a primary key. This field uniquely identifies each row in your table. It's crucial for efficient data retrieval and ensuring data integrity. Primary keys in SpaceTimeDB are typically `u64` (unsigned 64-bit integers).
- **Indexes (`#[index]`)**: While a primary key ensures unique identification, secondary indexes improve the performance of queries that filter or sort by other fields. If you frequently search for users by their `name` or items by their `category`, adding an `#[index]` to those fields will speed up those operations significantly.

- **Unique Constraints (`#[unique]`)**: You can also mark a field with `#[unique]` to ensure that no two rows in the table have the same value for that specific field. This is often used for things like usernames or email addresses.

Here's a conceptual view of how your Rust code maps to SpaceTimeDB tables:



Relations: Connecting Your Data

In SpaceTimeDB, you don't define explicit `FOREIGN KEY` constraints in the same way you would in a traditional SQL database. Instead, relationships between tables are established logically by referencing the primary key of one table in another.

For example, if you have a `User` table and a `TodoItem` table, a `TodoItem` doesn't "belong" to a `User` through a database constraint. Instead, the `TodoItem` struct would simply contain a field, say `user_id`, which stores the `id` of the `User` it's associated with.

This approach offers flexibility and is very efficient for real-time synchronization. SpaceTimeDB's client SDKs (which we'll cover later) make it easy to "join" this data on the client side, allowing you to build rich UIs that display related information.

Why this approach?

- **Flexibility:** Avoids rigid database-level constraints that can sometimes complicate schema evolution or distributed systems.
- **Performance:** Lookup by primary key (ID) is extremely fast.
- **Client-side Joins:** SpaceTimeDB's reactive nature means clients can efficiently subscribe to and combine data from multiple tables based on these logical links.

Step-by-Step Implementation: Building Our First Schema

Let's put these concepts into practice by creating a simple "Todo List" application schema. Our application will need to store `User` information and `TodoItem`s, with each `TodoItem` belonging to a specific `User`.

Prerequisites

Make sure you've completed the setup from Chapter 2 and have the `spacetime` CLI installed. We'll assume you've already created a new SpaceTimeDB project (e.g., `spacetime new todo-app`) and are working within its directory.

If you haven't, open your terminal and run:

```
spacetime new todo-app
cd todo-app
```

1. Open Your Schema File

Navigate to the `src` directory within your `todo-app` project. You'll find a file named `spacetime.rs`. This is where your schema and server-side logic will live.

Open `src/spacetime.rs` in your favorite code editor. It might contain some boilerplate code. For now, let's clear it out so we can start fresh.

Your `src/spacetime.rs` file should initially look something like this (you might remove any example structs if present):

```
#![allow(warnings)] // Allow warnings for now, good practice to remove later

use spacetimedb::{
    spacetimedb,
    SpacetimeTable,
};

// Your schema definitions and reducers will go here
```

2. Define the `User` Table

First, let's define our `User` table. Each user will have a unique `id` and a `name`.

Add the following Rust code to your `src/spacetime.rs` file, just below the `use` statements:

```
#[spacetimedb(table)]
pub struct User {
    #[primarykey]
    pub id: u64,
    pub name: String,
}
```

Let's break down these lines: * `#[spacetimedb(table)]`: This is the attribute that marks our `User` struct as a SpaceTimeDB table. It's a newer, more concise syntax that replaces `#[derive(SpacetimeTable)]` in recent `v2.x` versions of SpaceTimeDB, aligning with modern Rust attribute usage. * `pub struct User { ... }`: Defines a public Rust struct named `User`. `pub` means it's publicly accessible. * `#[primarykey]`: This attribute designates the `id` field as the primary key for the `User` table. It must be unique for each `User` entry. * `pub id: u64`: The `id` field is a public unsigned 64-bit integer, a common type for primary keys. * `pub name: String`: The `name` field is a public Rust `String`, which will store the user's name.

3. Define the `TodoItem` Table

Next, let's define the `TodoItem` table. Each todo item will have its own unique `id`, a `user_id` to link it to a `User`, a `description`, and a `completed` status.

Add this code below your `User` struct definition in `src/spacetime.rs`:

```
#[spacetimedb(table)]
pub struct TodoItem {
    #[primarykey]
    pub id: u64,
    #[index]
    pub user_id: u64, // Logical link to a User's id
    pub description: String,
    pub completed: bool,
}
```

And here's the explanation: * `#[spacetimedb(table)]`: Again, marking `TodoItem` as a SpaceTimeDB table. * `pub id: u64`: The primary key for the `TodoItem`. * `#[index] pub user_id: u64`: This field holds the `id` of the `User` who owns this todo item. We've added `#[index]` because we'll likely want to quickly retrieve all todo items for a specific user. This index will make those lookups fast. * `pub description: String`: The text content of the todo item. * `pub completed: bool`: A boolean indicating whether the todo item has been completed.

Your complete `src/spacetime.rs` file should now look like this:

```
#![allow(warnings)] // Allow warnings for now, good practice to remove later

use spacetimedb::{
    spacetimedb,
    SpacetimeTable,
};

#[spacetimedb(table)]
pub struct User {
    #[primarykey]
    pub id: u64,
    pub name: String,
}

#[spacetimedb(table)]
pub struct TodoItem {
    #[primarykey]
    pub id: u64,
    #[index]
    pub user_id: u64, // Logical link to a User's id
    pub description: String,
    pub completed: bool,
}
```

4. Compiling and Deploying Your Schema

Now that we've defined our schema, it's time to compile our Rust code into a WebAssembly module and then deploy it to a SpaceTimeDB instance.

1. **Build the Module**: Open your terminal in the `todo-app` project directory and run: `bash spacetime build` This command invokes the Rust compiler

to build your `src/spacetime.rs` file into a `.wasm` file, typically located at `target/spacetime.wasm`. You should see output indicating a successful build. If there are any Rust syntax errors, the compiler will tell you here!

2. **Deploy the Module**: Once built, deploy it to your local SpaceTimeDB instance. We'll use the `spacetime db start` command, which both starts the database and deploys your module.

   `bash spacetime db start --module-path target/spacetime.wasm` You should see output indicating that SpaceTimeDB is starting up and your module has been successfully deployed. This command effectively applies your schema to the running SpaceTimeDB instance.

   Note on `spacetime db deploy` vs `spacetime db start --module-path`: * `spacetime db deploy` is used to deploy a new module to an already running SpaceTimeDB instance. * `spacetime db start --module-path` is convenient for local development, as it starts the database and deploys your module in one go. We'll use this primarily for now.

5. Verifying Your Schema

How do we know our schema was deployed correctly? The SpaceTimeDB CLI provides ways to inspect the database.

While SpaceTimeDB `v2.x` focuses heavily on client SDKs for interaction, you can get a basic confirmation of table existence.

1. **Connect to the CLI (if not already connected):** If your `spacetime db start` command is running in one terminal, open a second terminal in the `todo-app` directory.

2. **List Tables (Conceptual):** Currently, direct CLI commands for listing schema details are being continuously enhanced. The primary way to interact and observe your schema is through client SDKs (which we'll cover in the next chapter) or by checking the logs of your `spacetime db start` command for successful module deployment.

   For now, trust that if `spacetime build` and `spacetime db start --module-path` completed without errors, your schema has been successfully applied! The SpaceTimeDB instance now understands the `User` and `TodoItem` tables and their defined fields.

   In future chapters, when we connect a client, we'll see exactly how to query and interact with these tables.

Mini-Challenge: Enhancing Your TodoItem

Let's expand our `TodoItem` table with a couple more practical fields. This will solidify your understanding of adding fields and applying indexes.

Challenge: 1. Add a `priority` field to the `TodoItem` struct. This field should be an `u8` (unsigned 8-bit integer), representing priority levels from 0 (low) to 255 (high). 2. Make the `priority` field indexable, as you might want to quickly find all high-priority tasks. 3. Add an `created_at` field to the `TodoItem` struct. This field should be a `u64`, storing a Unix timestamp (milliseconds since epoch) indicating when the todo item was created. This field does not need an index for now.

Hint: * Remember the `#[index]` attribute for indexable fields. * After making changes to `src/spacetime.rs`, you'll need to run `spacetime build` again, and then restart your database with `spacetime db start --module-path target/spacetime.wasm` to apply the new schema.

Click for Solution (after you've tried it!)

```
#![allow(warnings)]

use spacetimedb::{
    spacetimedb,
    SpacetimeTable,
};

#[spacetimedb(table)]
pub struct User {
    #[primarykey]
    pub id: u64,
    pub name: String,
}

#[spacetimedb(table)]
pub struct TodoItem {
    #[primarykey]
    pub id: u64,
    #[index]
    pub user_id: u64, // Logical link to a User's id
    pub description: String,
    pub completed: bool,
    #[index] // Added index here
    pub priority: u8, // New field for priority
    pub created_at: u64, // New field for creation timestamp
}
```

After updating the file, run:

```
spacetime build
spacetime db start --module-path target/spacetime.wasm
```

What to observe/learn: You've now successfully modified your schema, added new data types, and applied another index. This demonstrates the iterative nature of schema design in SpaceTimeDB.

Common Pitfalls & Troubleshooting

Even with a strong type system like Rust's, you might encounter a few common issues when designing your SpaceTimeDB schema.

1. **Rust Compilation Errors**: The most frequent issue for beginners.

- **Problem**: `spacetime build` fails with Rust compiler errors.
- **Solution**: Read the error messages carefully. Rust's compiler is famously helpful! It often tells you exactly where the error is, what's expected, and sometimes even suggests fixes. Common issues include missing semicolons, incorrect types, or uninitialized fields. 2. **Missing `#[spacetimedb(table)]` or `#[primarykey]`**:
- **Problem**: Your module builds, but SpaceTimeDB doesn't recognize your table, or deployment fails with a schema-related error.
- **Solution**: Double-check that every struct intended to be a table has `#[spacetimedb(table)]` above it, and that exactly one field within each table struct is marked with `#[primarykey]`. 3. **Incorrect Data Types**:
- **Problem**: You try to store a `String` in a `u64` field, or vice-versa, which will be caught by the Rust compiler.
- **Solution**: Ensure your Rust types match the kind of data you intend to store. SpaceTimeDB supports most primitive Rust types and certain complex types. Consult the official SpaceTimeDB documentation for a comprehensive list of supported types: https://spacetimedb.com/docs/ 4. **Forgetting `spacetime build` or `spacetime db start`**:
- **Problem**: You make changes to `spacetime.rs` but your database doesn't reflect them, or your client code (in later chapters) can't find the new fields/tables.
- **Solution**: Always remember the two-step process: `spacetime build` to compile your Rust code, then `spacetime db start --module-path`

`target/spacetime.wasm` (or `spacetime db deploy`) to apply the compiled module to the running database instance.

Summary: Your Data Takes Shape

Congratulations! You've successfully designed and deployed your first SpaceTimeDB schema. Let's recap the key takeaways from this chapter:

- **Schema as Blueprint**: Your SpaceTimeDB schema, written in Rust, defines both your data's structure and your server-side logic, compiled into a WebAssembly module.
- **Tables from Structs**: Rust `struct`s decorated with `#[spacetimedb(table)]` become your database tables.
- **Primary Keys**: Every table must have a `#[primarykey]` field (typically `u64`) for unique identification.
- **Indexes for Speed**: Use `#[index]` on fields you'll frequently query or filter by to ensure performant lookups. `#[unique]` ensures field values are distinct across rows.
- **Logical Relations**: Relationships between tables are established by referencing primary keys (e.g., `user_id` in `TodoItem` linking to `User.id`), rather than explicit foreign key constraints.
- **Build and Deploy**: The workflow involves `spacetime build` to compile your Rust module and `spacetime db start --module-path target/spacetime.wasm` (or `spacetime db deploy`) to apply it to your SpaceTimeDB instance.

You now have a solid understanding of how to structure your application's data within SpaceTimeDB. In the next chapter, we'll learn how to interact with this schema by writing server-side logic using SpaceTimeDB's "reducers" to create, update, and delete data, bringing your real-time application to life!

References

- SpacetimeDB Official Documentation
- SpacetimeDB GitHub Repository
- Rust Programming Language Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 12

Chapter 4: Querying Your Data: Retrieving and Filtering Information

Introduction

Welcome back, future SpaceTimeDB master! In the previous chapter, you learned how to define your database schema and create tables to store your application's shared state. You even got a taste of how to add data to these tables using reducers. But what good is storing data if you can't get it back out?

This chapter is all about **querying your data**. We'll dive into how clients can ask SpaceTimeDB for specific pieces of information and how that information is kept up-to-date in real-time. We'll explore the unique subscription model that makes SpaceTimeDB so powerful for real-time applications, and also touch upon how server-side logic (like your reducers) can access and filter data. By the end of this chapter, you'll be able to retrieve exactly the data you need, when you need it, and react to changes instantly.

Ready to make your data come alive? Let's go!

Core Concepts: How SpaceTimeDB Queries Data

SpaceTimeDB takes a slightly different approach to data retrieval compared to traditional request-response databases. While you can certainly "query" data in a familiar sense, its strength lies in **real-time subscriptions** where clients declare their interest in data and receive continuous updates.

The Two Sides of Querying

In SpaceTimeDB, querying happens in two primary contexts:

1. **Client-Side Subscriptions**: This is the primary way your frontend applications (web, game clients, mobile apps) retrieve data. Clients "subscribe" to tables or specific views of tables. Once subscribed, SpaceTimeDB streams the initial data and then pushes any subsequent changes to that data in real-time. This is perfect for building reactive UIs and multiplayer experiences.

2. **Server-Side Data Access (within Modules/Reducers)**: Your SpaceTimeDB modules, written in Rust, can also query the database. This is typically done within reducers or other server-side logic to read existing state before making modifications or performing complex calculations. This is similar to how a traditional backend service might query its database.

Let's explore each of these in more detail.

Client-Side Subscriptions: Your Window to Real-time Data

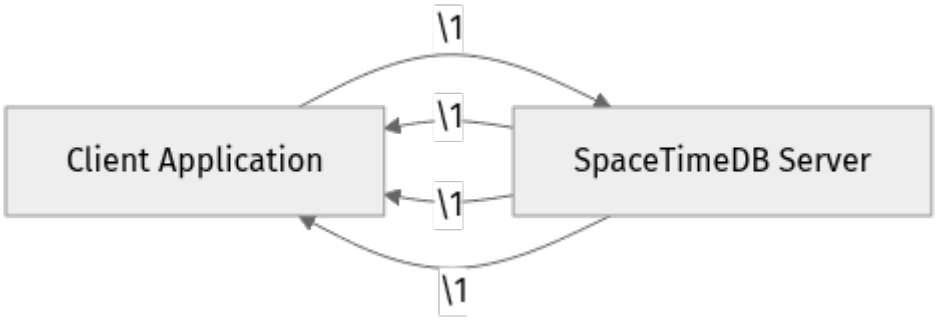
Imagine you're building a chat application. You don't just want to fetch all messages once; you want to see new messages as they arrive, instantly. That's where subscriptions shine!

A client-side subscription is a declaration of interest. Your client tells SpaceTimeDB, "Hey, I'd like to see all messages in this chat room," or "Show me all players currently online." SpaceTimeDB then does three things:

1. **Initial Snapshot**: Sends the client all the data that matches the subscription's criteria right now.
2. **Continuous Updates**: Whenever that data changes (e.g., a new message is sent, a player logs in/out, or a player's position updates), SpaceTimeDB automatically pushes those changes to your client.
3. **Filtered Views**: You can specify criteria (filters) to only receive a subset of the data, keeping your client's data footprint small and relevant.

This mechanism fundamentally changes how you build real-time applications, moving away from constant polling and towards an efficient, event-driven model.

Here's a simplified flow:



Server-Side Data Access: Logic Meets State

While client subscriptions are for broadcasting data to clients, your server-side SpaceTimeDB modules (the Rust code) often need to read data to perform their logic. For instance, a reducer that handles a `join_game` event might first need to

check how many players are already in the game before allowing a new one to join.

Within your Rust modules, you interact directly with the database tables you've defined. SpaceTimeDB provides a clear API for iterating through table rows, filtering them, and retrieving specific entries. This is where you'll use more traditional-looking query patterns, but still within the deterministic, event-sourced context of SpaceTimeDB.

The `spacetime` CLI for Data Exploration

Before we dive into code, remember your trusty `spacetime` CLI tool! It's not just for deploying modules; it's also fantastic for inspecting the current state of your database.

You can connect to your local or remote SpaceTimeDB instance and browse tables, view rows, and even manually insert data. This is invaluable for debugging and understanding what's actually stored.

Challenge for yourself: If you have your SpaceTimeDB instance running from Chapter 3, try connecting to it with `spacetime client` and then use commands like `list tables` or `select * from {your_table_name}` to see the data you inserted previously. This will give you a feel for interacting with the database directly.

Step-by-Step Implementation: Subscribing and Filtering

Let's put these concepts into practice. We'll continue with our simple `Player` table from Chapter 3 and learn how a client can subscribe to it and filter the results.

Prerequisites

Make sure you have: 1. A SpaceTimeDB project initialized (e.g., `my_game_db`). 2. A `Player` table defined in your `src/lib.rs` module, similar to this:

```rust
// src/lib.rs
use spacetimedb::{spacetimedb, ReducerContext, Identity, Timestamp};

#[spacetimedb(table)]
pub struct Player {
    #[primarykey]
    pub identity: Identity,
    pub name: String,
    pub health: u32,
    pub last_login: Timestamp,
}

#[spacetimedb(reducer)]
pub fn create_player(ctx: ReducerContext, name: String) {
    if Player::filter_by_identity(&ctx.sender).is_some() {
        log::info!("Player with identity {:?} already exists.", ctx.sender);
        return;
    }

    Player::insert(Player {
        identity: ctx.sender,
        name,
        health: 100,
        last_login: ctx.timestamp,
    }).expect("Failed to insert player");

    log::info!("Player {:?} created with name: {}", ctx.sender, name);
}
```

1. Your SpaceTimeDB instance running locally: `bash spacetime db dev --disable-component-hot-reloading`
2. Your module deployed: `bash spacetime client deploy .`
3. Some sample `Player` data. If you haven't inserted any, you can do so from the `spacetime client` console: `spacetime client // In the client console: create_player("Alice") create_player("Bob") create_player("Charlie") create_player("Alice_Alt") // A player with a similar name` Each `create_player` call will use a new ephemeral identity by default, creating unique players.

Step 1: Setting up a Basic Client

We'll use a simple JavaScript/TypeScript client for this example, as it's common for web and Node.js applications. Create a new file, `client.js` (or `client.ts` if you prefer TypeScript), in your project root.

First, install the SpaceTimeDB client library:

```
npm init -y
npm install @clockworklabs/spacetimedb-sdk
```

Now, open `client.js` and add the basic connection logic:

```
// client.js
import { SpacetimeDBClient } from "@clockworklabs/spacetimedb-sdk";

// Define your SpaceTimeDB instance URL.
// For local development, this is typically ws://localhost:3000
const SPACETIMEDB_URI = "ws://localhost:3000";

// Initialize the client
const client = new SpacetimeDBClient(SPACETIMEDB_URI);

console.log("Connecting to SpaceTimeDB...");

client.onConnect(() => {
    console.log("Successfully connected to SpaceTimeDB!");
    // We'll add our subscription logic here
});

client.onDisconnect(() => {
    console.log("Disconnected from SpaceTimeDB.");
});

client.onError((e) => {
    console.error("SpaceTimeDB client error:", e);
});

// Connect to the database
client.connect();
```

Run this client:

```
node client.js
```

You should see "Connecting to SpaceTimeDB..." and then "Successfully connected to SpaceTimeDB!". Great, your client can talk to the database!

Step 2: Subscribing to All Players

Let's subscribe to the `Player` table to get all player data. We'll also set up an event listener to react to data changes.

Modify your `client.js` file:

```
// client.js
import { SpacetimeDBClient } from "@clockworklabs/spacetimedb-sdk";
// Import your table definitions from the generated client library
// Assuming your module ID is 'my_game_db' and you've run 'spacetime client
generate'
// For now, we'll manually define the Player structure for logging clarity.
// In a real project, you'd import generated types.

const SPACETIMEDB_URI = "ws://localhost:3000";
const MODULE_NAME = "my_game_db"; // Replace with your actual module name

const client = new SpacetimeDBClient(SPACETIMEDB_URI);

// We'll simulate the generated Player class for this example's logging
class Player {
    constructor(identity, name, health, last_login) {
        this.identity = identity;
        this.name = name;
        this.health = health;
        this.last_login = last_login;
    }
}

// Map of table names to their data
const subscribedData = {};

client.onConnect(() => {
    console.log("Successfully connected to SpaceTimeDB!");

    // Subscribe to the 'Player' table
    client.subscribe([`/${MODULE_NAME}/Player`]); // Subscribe to the full
table

    console.log("Subscribed to the Player table.");
});

// Listen for updates to subscribed data
client.onUpdate(() => {
    console.log("\n--- Data Update Received ---");
    // Get all rows for the 'Player' table
    const players = client.getEntities(MODULE_NAME, "Player");

    // Clear previous data for this example (in a real app, you'd manage state)
    subscribedData["Player"] = [];

    if (players && players.length > 0) {
        console.log("Current Players:");
        players.forEach(playerRow => {
            // In a real app, 'playerRow' would be an instance of your
generated Player class
            // For this example, we'll construct it for consistent logging.
            const player = new Player(

playerRow.identity.toHexString(), // Convert Identity to string for display
                playerRow.name,
                playerRow.health,
                playerRow.last_login
            );
            subscribedData["Player"].push(player);
            console.log(`- Name: ${player.name}, Health: ${player.health}, ID:
${player.identity.substring(0, 8)}...`);
```

```
        });
    } else {
        console.log("No players found.");
    }
    console.log("----------------------------");
});


client.onDisconnect(() => {
    console.log("Disconnected from SpaceTimeDB.");
});

client.onError((e) => {
    console.error("SpaceTimeDB client error:", e);
});

client.connect();
```

Explanation of changes: * `client.subscribe([`/${MODULE_NAME}/Player`])`: This is the core of the subscription. We're telling SpaceTimeDB we want to receive data for the `Player` table within our `my_game_db` module. The array allows subscribing to multiple tables or views. * `client.onUpdate(() => { ... })`: This callback fires whenever any subscribed data changes. Inside it, we use `client.getEntities(MODULE_NAME, "Player")` to retrieve the current snapshot of all rows in the `Player` table that match our subscription. * `playerRow.identity.toHexString()`: The `Identity` type from SpaceTimeDB is a special object; we convert it to a hex string for easier display.

Run this updated `client.js` again. You should see an initial list of all players you created.

Now for the fun part: keep `client.js` running. Open a new terminal and connect to your SpaceTimeDB instance with the CLI:

```
spacetime client
```

In the CLI, call your `create_player` reducer:

```
create_player("Eve")
```

Observe your `client.js` terminal. You should immediately see a new "Data Update Received" log, and "Eve" will appear in the list of players! This demonstrates the real-time nature of subscriptions.

Step 3: Filtering Subscriptions

Subscribing to all data is often inefficient. What if we only want players with a certain name, or those with low health? SpaceTimeDB subscriptions allow you to add filters.

Let's modify our `client.js` to only subscribe to players named "Alice".

```
// client.js
import { SpacetimeDBClient } from "@clockworklabs/spacetimedb-sdk";

const SPACETIMEDB_URI = "ws://localhost:3000";
const MODULE_NAME = "my_game_db";

const client = new SpacetimeDBClient(SPACETIMEDB_URI);

class Player { /* ... same as before ... */ }
const subscribedData = {};

client.onConnect(() => {
    console.log("Successfully connected to SpaceTimeDB!");

    // --- NEW: Filtered Subscription ---
    // Subscribe to the 'Player' table, but only for rows where 'name' is
'Alice'
    client.subscribe([`/${MODULE_NAME}/Player?name=Alice`]);

    console.log("Subscribed to Player table, filtered by name='Alice'.");
});

// ... onUpdate, onDisconnect, onError, connect functions remain the same ...
// (The onUpdate logic will now only receive and log players named Alice)
```

Explanation of filter: * `/${MODULE_NAME}/Player?name=Alice`: Notice the `?name=Alice` at the end of the path. This is how you apply a simple equality filter directly in the subscription path. SpaceTimeDB's client SDK understands this query string syntax.

Run this modified `client.js`. You should now only see players named "Alice" (and "Alice_Alt" might not appear if the filter is strict equality, depending on how `name=Alice` is interpreted by the SDK against your schema - typically it's strict equality).

Experiment with other filters (stop and restart `client.js` each time): * `?health=100`: To see players with full health. * `?name=Bob`: To see only Bob.

SpaceTimeDB's client SDKs support various filter types, including:

- **Equality**: `?field=value`
- **Inequality**: `?field!=value`
- **Range**: `?field>=value`, `?field<=value`, `?field>value`, `?field<value`

- **Logical AND**: Combine multiple filters with `&` (e.g., `?name=Alice&health=100`)
- **Logical OR**: For more complex OR conditions, you might need to subscribe to multiple filtered views or filter on the client side after a broader subscription (depending on SDK capabilities and performance needs).

For the most up-to-date and comprehensive filtering options, always refer to the official SpaceTimeDB client SDK documentation.

Step 4: Querying from a Reducer (Server-Side)

Now, let's look at how your Rust modules can access data. This is crucial for implementing game logic, validation, or complex state transitions.

Imagine we want a reducer that "heals" a player but only if their health is below a certain threshold. This requires reading the player's current health.

Open your `src/lib.rs` file and add a new reducer:

```
// src/lib.rs
// ... existing code ...

#[spacetimedb(reducer)]
pub fn heal_player(ctx: ReducerContext, amount: u32) {
    // 1. Retrieve the player using the sender's identity
    let mut player = match Player::filter_by_identity(&ctx.sender) {
        Some(p) => p,
        None => {
            log::warn!("Heal attempt by non-existent player: {:?}", ctx.sender)
;
            return;
        }
    };

    // 2. Access the player's current health
    let current_health = player.health;
    log::info!("Player {} (ID: {:?}) current health: {}", player.name, player.i
dentity, current_health);

    // 3. Apply game logic: Only heal if not already at max health (e.g., 100)
    if current_health >= 100 {
        log::info!("Player {} is already at max health.", player.name);
        return;
    }

    // 4. Calculate new health, capping at 100
    player.health = u32::min(current_health + amount, 100);

    // 5. Update the player in the database
    // The `update` method takes a closure that receives the current row
    // and returns the modified row.
    player.update().expect("Failed to update player health");

    log::info!("Player {} (ID: {:?}) healed by {} to {} health.", player.name,
player.identity, amount, player.health);
}
```

Explanation: * `Player::filter_by_identity(&ctx.sender)`: This is how you query a table by its primary key (which is `identity` in our `Player` table). It returns an `Option<Player>`, so we use a `match` statement to handle both `Some` (player found) and `None` (player not found) cases. * `player.health`: Once you have a `Player` instance, you can directly access its fields. * `player.update()`: After modifying the `player` struct, call `.update()` to persist the changes back to the database. This will trigger real-time updates for any clients subscribed to this player's data!

Deploy and Test: 1. Save `src/lib.rs`. 2. Redeploy your module: `bash spacetime client deploy .` 3. Keep your `client.js` running (subscribed to all players, or specifically to "Alice" if you want to observe her health). 4. In the `spacetime client` console, assume the identity of "Alice" (or any other player you created) and then call `heal_player`: `login_with_identity "0x..." // Replace with Alice's identity heal_player(10)` You'll need to know the

identity of one of your players. You can find this by running `spacetime client` and using `select * from my_game_db.Player` to see the `identity` column.

```
You should see the `heal_player` reducer's logs in your `spacetime db dev`
terminal, and crucially, your `client.js` terminal will show a "Data Update
Received" with Alice's new health!
```

This demonstrates how reducers can perform reads, apply logic, and then write updates, all while maintaining real-time synchronization with connected clients.

Mini-Challenge: Filtering Items by Type

Let's solidify your understanding with a practical challenge.

Challenge: 1. **Define a new table** called `Item` in your `src/lib.rs`. It should have a `#[primarykey] id: u64`, `name: String`, `item_type: String` (e.g., "weapon", "armor", "potion"), and `rarity: String` (e.g., "common", "rare", "legendary"). 2. **Create a reducer** called `create_item` that allows you to insert new items into this table. 3. **Deploy** your updated module. 4. **Insert at least 5-7 sample items** with varying `item_type` and `rarity` using the `spacetime client` CLI. 5. **Modify your `client.js`** to: * Subscribe to the `Item` table. * **Filter this subscription** to only receive items where `item_type` is "weapon" AND `rarity` is "legendary". * Log the details of the items received, just like you did for players.

Hint: * Remember the `?field=value&another_field=another_value` syntax for combining filters in client subscriptions. * For the `id` field in your `create_item` reducer, you can use `spacetimedb::random_id()` to generate unique IDs.

What to Observe/Learn: * How to define a new table and reducer independently. * How to combine multiple filters in a single client-side subscription. * The immediate real-time updates when you add new items matching your filter criteria.

Common Pitfalls & Troubleshooting

1. **Incorrect Subscription Path**:

- **Mistake**: Using `/Player` instead of `/{MODULE_NAME}/Player`. For example, if your module ID is `my_game_db`, it should be `/my_game_db/Player`.

- **Troubleshooting**: Double-check your `MODULE_NAME` constant and ensure it matches the ID specified in your `Spacetime.toml` or the ID you see when deploying. Look for client-side errors indicating an invalid subscription.

1. **Forgetting `client.onUpdate`**:

- **Mistake**: You called `client.subscribe()`, but your `onUpdate` callback never fires, or you're not correctly retrieving data inside it.
- **Troubleshooting**: Ensure `client.onUpdate()` is registered before `client.connect()`. Inside `onUpdate`, verify that `client.getEntities(MODULE_NAME, "TableName")` is being called and that the table name is correct. Remember that `onUpdate` fires for any change to any subscribed table, so you might need to check which table changed if you have multiple subscriptions.

1. **Misunderstanding Filter Syntax**:

- **Mistake**: Using incorrect query parameters for filtering (e.g., `?name==Alice` instead of `?name=Alice`, or `?health>50` when the SDK expects a different range syntax).
- **Troubleshooting**: Refer to the official SpaceTimeDB client SDK documentation for the exact filter syntax supported by your chosen client library. Different SDKs might have slightly different ways to express complex queries.

1. **No Data Appearing**:

- **Mistake**: Your client connects, subscribes, but no data ever shows up.
- **Troubleshooting**: * Is your SpaceTimeDB instance running? (`spacetime db dev`) * Is your module deployed? (`spacetime client deploy .`) * Have you inserted any data into the table you're subscribing to? Use `spacetime client` and `select * from {your_module_name}.{your_table_name}` to verify data exists on the server. * Is your client connected to the correct URI (`ws://localhost:3000` for local dev)?

Summary

Phew! You've just unlocked a crucial part of building real-time applications with SpaceTimeDB. Here's a quick recap of what we covered:

- **Client-Side Subscriptions** are the backbone of real-time data flow in SpaceTimeDB, allowing clients to declare interest in data and receive continuous updates.
- You learned how to use `client.subscribe()` with **path-based filters** (e.g., `?name=Alice`) to retrieve specific subsets of data.
- The `client.onUpdate()` callback is your entry point for reacting to any changes in your subscribed data.
- **Server-Side Data Access** within Rust reducers allows you to read table data (e.g., `Player::filter_by_identity()`) to inform your application logic before making state changes.
- The `spacetime client` CLI is an invaluable tool for inspecting your database's current state.

You now have the tools to retrieve and filter data effectively, both from your client applications and within your server-side logic. This is a massive step towards building dynamic, reactive systems.

In the next chapter, we'll shift our focus from reading data to **modifying data**. You'll learn how to use reducers to change existing records, delete entries, and keep your shared state perfectly synchronized across all clients.

References

- SpacetimeDB Official Documentation
- SpacetimeDB Client SDK Documentation
- SpacetimeDB GitHub Repository

CHAPTER 13

Chapter 5: Bringing Logic to Life: Reducers and Server-Side Operations

Introduction: Where the Magic Happens – Server-Side Logic

Welcome back, intrepid SpaceTimeDB explorer! In our previous chapters, we laid the groundwork by understanding SpaceTimeDB's unique architecture, setting up our environment, and defining our database schema with tables. You now know how to structure your data, but what about changing it? How do you update a player's score, add a new chat message, or move a character in a game?

This is where server-side logic comes into play, and in SpaceTimeDB, it's handled by a powerful concept called **Reducers**. Reducers are the heart of your application's state changes, ensuring that all modifications to your shared database are consistent, deterministic, and immediately propagated to all connected clients.

In this chapter, we're going to dive deep into: * What Reducers are and why they are fundamental to SpaceTimeDB. * How Reducers enable event-driven updates and real-time synchronization. * The practical steps to write your first Reducer using Rust. * How clients interact with these server-side functions.

By the end of this chapter, you'll not only understand how to bring your application's logic to life but also appreciate the elegant simplicity SpaceTimeDB offers for building complex real-time systems. Get ready to write some Rust code and see your database react in real-time!

Core Concepts: The Power of Reducers

Imagine you have a shared whiteboard, and many people are trying to draw on it at the same time. Without rules, it would be chaos! Reducers are like the strict, fair rules for drawing on SpaceTimeDB's global whiteboard – they dictate exactly how and when changes can be made, ensuring everyone sees the same, consistent picture.

What is a Reducer?

At its core, a **Reducer** in SpaceTimeDB is a **deterministic function** that lives and executes on the SpaceTimeDB server. Its sole purpose is to receive inputs from a client and, based on those inputs, perform operations that modify the shared database state.

Think of it this way:

- **Client-Side:** Your frontend (web app, game, mobile) doesn't directly write to the database. Instead, it calls a Reducer function on the SpaceTimeDB server.
- **Server-Side:** The SpaceTimeDB server receives the call, executes the corresponding Reducer (which you write in Rust), and if the Reducer successfully modifies data, SpaceTimeDB automatically broadcasts those changes to all subscribed clients in real-time.

This architecture has profound implications for building real-time, collaborative, and multiplayer applications:

1. **Consistency:** All state changes are processed centrally and atomically by the server. This eliminates race conditions and ensures that every client sees the exact same, correct state.
2. **Determinism:** Reducers must be deterministic. This means that given the same inputs, a Reducer will always produce the same output state change. This is crucial for SpaceTimeDB's internal mechanics, such as replication, fault tolerance, and even for enabling features like time-travel debugging.
3. **Real-time Synchronization:** After a Reducer successfully modifies the database, SpaceTimeDB automatically pushes these updates to all relevant connected clients. You don't need to write any explicit synchronization or pub/sub code!

Reducers vs. Traditional Backend Endpoints

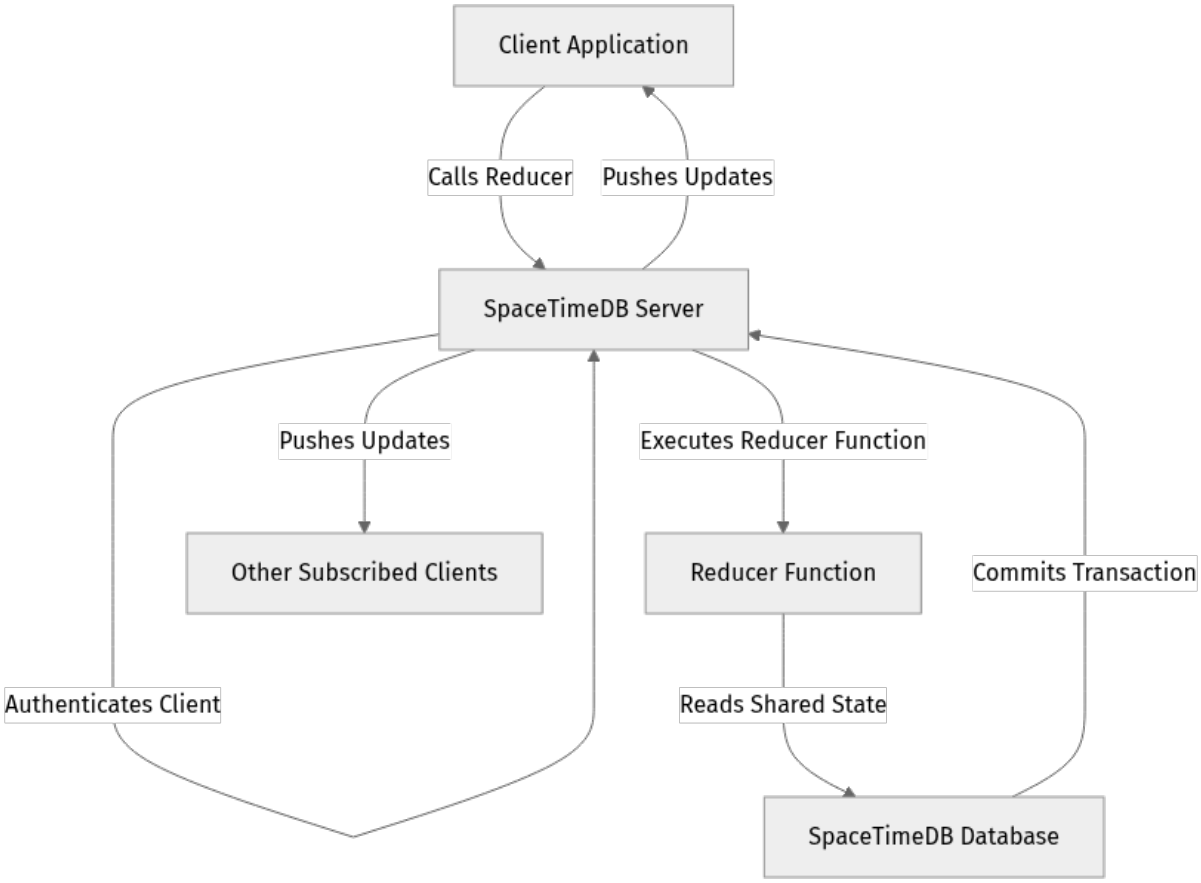
In a traditional backend, you might have REST API endpoints (e.g., `/api/items POST`) or GraphQL mutations. These endpoints typically involve: 1. Receiving an HTTP request. 2. Authenticating and authorizing the user. 3. Performing business logic. 4. Interacting with a database (e.g., SQL, NoSQL). 5. Optionally, broadcasting updates via WebSockets to other clients.

SpaceTimeDB's Reducers encapsulate steps 2-5 into a single, unified, and highly optimized system. Your Rust Reducer code handles the business logic and database interaction directly, and SpaceTimeDB takes care of the real-time

propagation automatically. This significantly streamlines development for real-time applications.

The Reducer Workflow

Let's visualize the journey of a Reducer call:



1. **Client Initiates:** A client application (e.g., a JavaScript frontend) calls a named reducer function, passing any necessary arguments.
2. **Server Receives & Validates:** The SpaceTimeDB server receives the call. It verifies the client's identity and checks if they have permission to call that reducer.
3. **Reducer Execution:** The server executes the corresponding Rust function you've defined as a reducer. This function contains your business logic.
4. **State Modification:** Inside the reducer, you can read from and write to your SpaceTimeDB tables using helper functions provided by the `spacetimedb` crate.
5. **Commit & Propagation:** If the reducer executes successfully, SpaceTimeDB commits the changes to the database. Crucially, it then intelligently identifies which clients are subscribed to the affected data and

immediately pushes the updates to them over the persistent WebSocket connection.

6. **Client Updates:** Clients automatically receive these updates, allowing their local state and UI to reflect the changes in real-time without polling or manual refresh.

Reducer Structure in Rust

Reducers are written in Rust and marked with the `#[reducer]` attribute macro. They typically take an `Identity` (representing the calling client) and custom arguments, and return a `Result<(), String>` to indicate success or failure.

Here's a sneak peek at what a reducer might look like:

```
use spacetimedb::{spacetimedb, table, reducer, Identity, Effect, Hash};

// (Assuming a 'Counter' table is defined in module.stdb)

#[reducer]
pub fn increment_counter(identity: Identity, counter_id: u32, amount: u32) -> R
esult<(), String> {
    // ... logic to find and update the counter ...
    Ok(()) // Indicate success
}
```

The `Identity` argument is a powerful feature that provides information about the client making the call. This is essential for implementing authorization logic (e.g., "only the owner can modify this item"). We'll explore authentication and authorization in more detail in a later chapter.

Step-by-Step Implementation: Building Our First Reducer

Let's get practical! We'll create a simple counter application where clients can increment a shared counter. This will demonstrate how to define a table, write a reducer, deploy it, and call it from a client.

Prerequisites: * You have the `spacetime` CLI installed (latest stable version, `v2.x` as of 2026-03-14). * You have a SpaceTimeDB project initialized (e.g., `spacetime new my_counter_app`). * You are familiar with basic Rust syntax.

Step 1: Define Your Table Schema

First, we need a table to hold our counter's value. We'll define a `Counter` table in our `module.stdb` file.

Open your `module.stdb` file (typically located at the root of your SpaceTimeDB project) and add the following schema definition:

```
// module.stdb
#[table]
pub struct Counter {
    #[primarykey]
    pub id: u32,
    pub value: u32,
}
```

Explanation: * `#[table]`: This attribute macro marks the `Counter` struct as a SpaceTimeDB table. * `pub struct Counter`: Defines a public struct named `Counter`. * `#[primarykey]`: The `id` field is marked as the primary key, meaning each counter will have a unique `id`. * `pub id: u32`: A public field `id` of type `u32` (unsigned 32-bit integer). * `pub value: u32`: A public field `value` of type `u32` to store our counter's current value.

Save this file.

Step 2: Implement the `increment_counter` Reducer

Now, let's write the Rust code for our reducer. Open your `src/lib.rs` file within your SpaceTimeDB module (e.g., `my_counter_app/src/lib.rs`).

Add the following Rust code:

```
// src/lib.rs
use spacetimedb::{spacetimedb, table, reducer, Identity, Effect, Hash};
// Import the Counter table definition from module.stdb
// (SpacetimeDB automatically makes tables defined in module.stdb available)
// You might need to add `use crate::module_stdb::Counter;` if not auto-
imported,
// but often it's directly accessible depending on project structure.
// For simplicity, we'll assume direct access here.

// If you defined the table in module.stdb, SpaceTimeDB's build process
// generates a `module_stdb.rs` file. You might need to explicitly import it:
// use crate::module_stdb::{Counter, CounterTable}; // Adjust based on
generated module

// For this example, we'll rely on the `spacetimedb` crate's helpers
// which often abstract away the direct import of the generated table struct.

// A reducer that initializes a new counter
#[reducer]
pub fn create_counter(identity: Identity, initial_value: u32) -> Result<(), Str
ing> {
    // Check if a counter with id 1 already exists (for simplicity, we'll use
id 1)
    if Counter::filter_by_id(1).is_some() {
        return Err("Counter with ID 1 already exists.".to_string());
    }

    // Insert a new counter
    Counter::insert(Counter {
        id: 1, // Using a fixed ID for this simple example
        value: initial_value,
    });

    Effect::log("counter_created", &format!("New counter created by {:?} with
initial value: {}", identity, initial_value));
    Ok(())
}


// Our main reducer: incrementing the counter
#[reducer]
pub fn increment_counter(identity: Identity, counter_id: u32, amount: u32) -> R
esult<(), String> {
    // 1. Find the counter in the database
    let mut counter = Counter::filter_by_id(counter_id)
        .ok_or_else(|| format!("Counter with ID {} not found", counter_id))?;

    // 2. Perform the business logic: increment its value
    counter.value += amount;

    // 3. Update the table with the new counter value
    Counter::update_by_id(counter_id, counter);

    // Optional: Log an event for observability/auditing
    Effect::log("counter_incremented", &format!("Counter ID {} incremented by
{} by {:?}", counter_id, amount, identity));

    Ok(()) // Indicate that the reducer executed successfully
}

// You can add more reducers here later...
```

Explanation of the `increment_counter` reducer: * `use spacetimedb::...`: Imports necessary components from the `spacetimedb` crate. * `#[reducer]`: This macro transforms our Rust function into a SpaceTimeDB reducer. * `pub fn increment_counter(identity: Identity, counter_id: u32, amount: u32) -> Result<(), String>`: * `identity: Identity`: This argument is automatically provided by SpaceTimeDB and represents the authenticated caller. It's crucial for security and auditing. * `counter_id: u32`: The ID of the counter we want to increment. This is an argument passed by the client. * `amount: u32`: The value by which to increment the counter. Also passed by the client. * `-> Result<(), String>`: Reducers typically return `Result<(), String>`. `Ok(())` signifies success, while `Err("Some error message".to_string())` indicates failure. * `let mut counter = Counter::filter_by_id(counter_id)...`: This line uses a generated helper function (`filter_by_id`) to retrieve a `Counter` instance from the database based on its primary key. `ok_or_else` handles the case where the counter isn't found, returning an `Err`. * `counter.value += amount;`: This is our core business logic – incrementing the counter's value. * `Counter::update_by_id(counter_id, counter);`: This generated helper function updates the existing counter in the database with the modified `counter` struct. * `Effect::log(...)`: This is a helper for logging messages to the SpaceTimeDB server's console/logs. It's great for debugging and understanding reducer execution. * `Ok(())`: If everything goes well, we return `Ok(())`.

Step 3: Deploy Your SpaceTimeDB Module

Now that we've defined our schema and written our reducer, it's time to deploy it to our local SpaceTimeDB instance.

1. **Start your SpaceTimeDB server (if not already running):** `bash spacetime start` This will typically start a local SpaceTimeDB instance, often on `localhost:3000`.
2. **Deploy your module:** Navigate to your project's root directory in your terminal and run: `bash spacetime deploy` This command compiles your Rust code, bundles your schema, and pushes it to your running SpaceTimeDB instance. You should see output indicating a successful deployment, including the deployed module hash.

   If you encounter compilation errors, carefully check your Rust code for typos or syntax issues.

Step 4: Interact with the Reducer from a Client

Finally, let's call our reducer from a client application. For simplicity, we'll use the `spacetime` CLI's `call` command, which acts as a client. In a real application, you'd use a client SDK (e.g., TypeScript, Python, C#).

1. **First, create the counter:** We need to ensure a counter with `id: 1` exists before we can increment it. `bash spacetime call create_counter 1 0` This calls the `create_counter` reducer with `initial_value: 0`. You should see `Ok(())` if successful.
2. **Now, call the `increment_counter` reducer:** `bash spacetime call increment_counter 1 1` This calls our `increment_counter` reducer, targeting `counter_id: 1` and incrementing it by `amount: 1`.

   You should see `Ok(())` as output.
3. **Verify the change:** To see if the counter actually updated, you can query the table: `bash spacetime get Counter` You should see output similar to: `json [ { "id": 1, "value": 1 } ]` Run `spacetime call increment_counter 1 1` a few more times, and then `spacetime get Counter` again. You'll see the `value` increasing in real-time!

   In a real client application, you would subscribe to the `Counter` table, and your UI would automatically update without needing to manually `get` the data.

Mini-Challenge: Resetting the Counter

You've successfully built and called your first SpaceTimeDB reducer! Now, let's solidify that knowledge with a small challenge.

Challenge: Implement a new reducer named `reset_counter` that takes a `counter_id` as input and sets the `value` of that specific counter back to `0`.

Steps: 1. Open your `src/lib.rs` file. 2. Define a new public function marked with `#[reducer]`. 3. It should accept `identity: Identity` and `counter_id: u32`. 4. Inside the reducer, find the counter by its `id`. 5. If found, set its `value` field to `0`. 6. Update the counter in the database. 7. Return `Ok(())` on success, or an `Err` with a descriptive message if the counter is not found. 8. Deploy your updated module using `spacetime deploy`. 9. Test your new reducer using `spacetime call reset_counter 1` (assuming your counter ID is 1). 10. Verify the result with `spacetime get Counter`.

Hint: The structure will be very similar to `increment_counter`. Remember to handle the case where the `counter_id` might not exist.

Common Pitfalls & Troubleshooting

Working with server-side logic can sometimes introduce new challenges. Here are a few common pitfalls and how to troubleshoot them:

1. **Reducer Not Found / Mismatched Signature:**

- **Symptom:** Your client call fails with an error like "Reducer `my_reducer` not found" or "Incorrect number/type of arguments".
- **Cause:** * You haven't deployed the module after adding/changing the reducer. * The reducer name you're calling from the client doesn't exactly match the Rust function name. * The number or types of arguments passed from the client don't match the reducer's function signature in Rust.
- **Fix:** * Always run `spacetime deploy` after making changes to `module.stdb` or `src/lib.rs`. * Double-check reducer names and argument types/counts. SpaceTimeDB's client SDKs often provide type-safe bindings, which help prevent this.

1. **Deterministic Errors in Reducers:**

- **Symptom:** Reducer execution fails unpredictably, or replicated instances of SpaceTimeDB diverge.
- **Cause:** You've introduced non-deterministic operations within your reducer. Examples include: * Generating random numbers (`rand::random()`). * Accessing the current wall-clock time (`std::time::Instant::now()`). * Making external network requests directly (e.g., HTTP calls to another service).
- **Fix:** Reducers must be deterministic. For operations like generating IDs, use `Hash::random()` (which is deterministically seeded by SpaceTimeDB's internal state) or sequential IDs. For time, pass a timestamp from the client if it's external, or use SpaceTimeDB's internal logical clock if available (advanced). External operations should be handled by separate "effect" services that react to SpaceTimeDB events, not directly within reducers.

1. **Permission Denied Errors:**

- **Symptom:** Your reducer is found, but the client call is rejected with a "Permission Denied" error.

- **Cause:** You (or the default setup) have implemented authorization logic that prevents the calling `Identity` from executing that specific reducer or modifying certain data.
- **Fix:** This is often a feature, not a bug! In later chapters, we'll learn about defining permissions. For now, if you're experimenting, ensure your default permissions allow `Identity::nil()` (the anonymous identity) to call your reducer, or use an authenticated client.

1. **Debugging Reducer Logic:**

- **Symptom:** Your reducer executes, but the database state isn't what you expect, or it returns an `Err`.
- **Fix:**
- **Use `Effect::log!`:** As shown in the example, `Effect::log!` is your best friend. You can print variable values, messages, and execution paths directly to the SpaceTimeDB server's console/logs.
- **Rust Debugging:** Since reducers are Rust code, you can use standard Rust debugging techniques (e.g., `dbg!`, `println!`, though `Effect::log!` is preferred for server-side output).
- **`spacetime get`:** Regularly use `spacetime get <TableName>` to inspect the current state of your tables after reducer calls.

Summary

Phew! You've just taken a massive leap forward in understanding SpaceTimeDB. Here's a quick recap of the key takeaways from this chapter:

- **Reducers are SpaceTimeDB's server-side logic:** They are deterministic Rust functions that live on the SpaceTimeDB server and are the only way to modify the database state.
- **Ensuring Consistency and Real-time:** Reducers guarantee atomic, consistent state changes that are automatically propagated to all subscribed clients in real-time.
- **Simplified Backend:** They unify database operations, business logic, and real-time synchronization, eliminating the need for separate API layers for data mutations.
- **Implementation:** You define tables in `module.stdb` and write your reducer functions in Rust, marking them with `#[reducer]`.

- **Deployment:** Use `spacetime deploy` to compile and push your Rust logic to the running SpaceTimeDB instance.
- **Client Interaction:** Clients call reducers by name, passing arguments, and SpaceTimeDB handles the rest, including real-time updates.
- **Determinism is Key:** Avoid non-deterministic operations within reducers to maintain consistency and enable SpaceTimeDB's advanced features.

You now have the fundamental building blocks to create interactive, real-time applications. You can define your data, and crucially, you can define how that data changes in a safe and consistent manner.

What's Next?

In the next chapter, we'll explore how to connect a frontend application (e.g., a web app) to SpaceTimeDB, subscribe to table changes, and truly see the "real-time" aspect come alive as your UI updates dynamically based on the reducer calls you've just learned to create. Get ready to build your first fully interactive SpaceTimeDB application!

References

- SpacetimeDB Official Website
- SpacetimeDB Documentation - Core Concepts
- SpacetimeDB Documentation - Reducers
- The Rust Programming Language Book

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 14

Chapter 6: Real-time Magic: Client Synchronization and Event Propagation

Chapter 6: Real-time Magic: Client Synchronization and Event Propagation

Welcome back, intrepid developer! In our previous chapters, we've explored the foundational concepts of SpaceTimeDB, from setting up your development environment to designing schemas and writing server-side logic using reducers. We've seen how SpaceTimeDB acts as a unified backend, combining a database with application logic.

Now, it's time to unveil the "magic" that makes SpaceTimeDB truly shine: its real-time capabilities. This chapter will pull back the curtain on how client applications stay perfectly synchronized with your SpaceTimeDB instance, receiving instant updates as data changes. We'll explore the core mechanisms of client synchronization, event propagation, and how to build responsive, collaborative experiences.

By the end of this chapter, you'll understand: * How SpaceTimeDB clients establish and maintain real-time connections. * The power of "subscriptions" to declaratively express interest in data. * How SpaceTimeDB efficiently propagates changes and events to connected clients. * How to integrate a basic SpaceTimeDB client into a web application. * The fundamental patterns for building real-time features like live dashboards or multiplayer game elements.

Ready to make your applications come alive with real-time updates? Let's dive in!

Core Concepts: The Pulse of Real-time

Traditional web applications often rely on a "request-response" model. A client asks for data, the server responds, and then the connection closes. To get updates, the client has to ask again (polling) or use complex, separate technologies like WebSockets and pub/sub systems. SpaceTimeDB elegantly solves this by integrating real-time synchronization directly into its core.

Client-Server Communication: Always Connected

At the heart of SpaceTimeDB's real-time capabilities is a persistent, bidirectional communication channel between the client and the SpaceTimeDB instance. This channel is typically established using **WebSockets**.

When a client connects, it doesn't just make a one-off request; it opens a continuous pipeline. This pipeline allows the client to: 1. **Send requests:** Call reducers, execute queries. 2. **Receive updates:** Get notified instantly when data it cares about changes on the server.

Think of it like a phone call versus sending a letter. With a phone call, once connected, you can have a continuous conversation, sending and receiving information in real-time.

The Power of Subscriptions

How does SpaceTimeDB know what data a client "cares about"? This is where **subscriptions** come into play. A subscription is a declarative instruction from the client to SpaceTimeDB, saying, "Hey, I'm interested in all data from `TableA`," or "I want to know about all users with `status = 'online'`."

Once subscribed, SpaceTimeDB continuously monitors the requested data. Any time a relevant change occurs (an insert, update, or delete), SpaceTimeDB automatically pushes that change to the subscribing client.

This is a stark contrast to traditional database queries, which are "point-in-time" snapshots. A subscription is a "live query" that continuously updates its results.

Why Subscriptions are Smart:

- **Efficiency:** Clients only receive data they explicitly ask for, reducing network traffic.
- **Simplicity:** Developers don't need to write complex polling logic or manage manual WebSocket messages. SpaceTimeDB handles the "when to send" and "what to send."
- **Consistency:** All subscribed clients see the same, consistent state as it evolves, thanks to SpaceTimeDB's deterministic nature.

How SpaceTimeDB Propagates Events

When a reducer executes and modifies the database, SpaceTimeDB doesn't just update its internal state; it also generates a stream of **events**. These events describe what changed (e.g., "row inserted into `users` table," "value updated in `messages` table").

SpaceTimeDB then intelligently compares these changes against all active client subscriptions. If a change affects data that a client is subscribed to, SpaceTimeDB serializes the relevant event and pushes it down the client's WebSocket connection.

This process is incredibly fast and efficient, designed to deliver updates with minimal latency. It's the engine that powers real-time collaboration and dynamic user interfaces.

Deterministic State and Event Sourcing

A key architectural principle of SpaceTimeDB (which we touched upon in previous chapters) is its foundation in **deterministic event sourcing**. Every change to the database is a result of a reducer executing based on an incoming client call. These reducer calls are logged as a sequence of events.

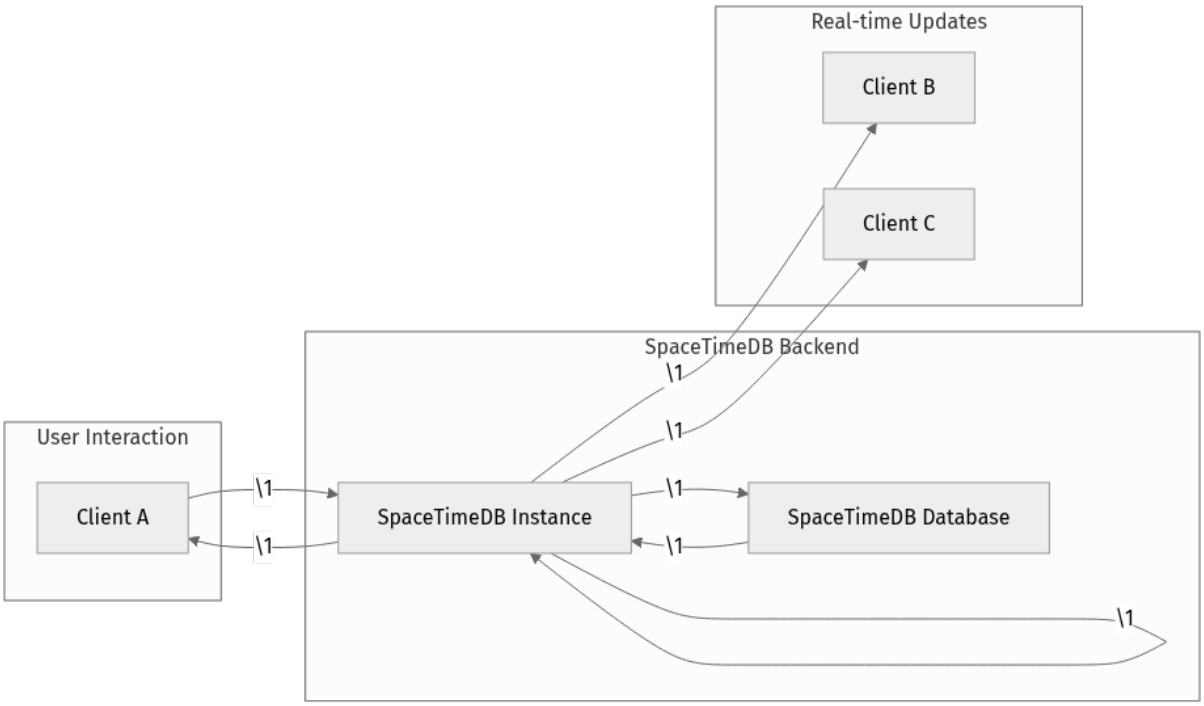
Because reducers are deterministic, applying the same sequence of events to an initial state will always result in the same final state. This property is crucial for:

- **Consistency:** Ensuring all clients, when caught up, see the identical database state.
- **Replication:** Easily replicating the database by replaying event logs.
- **Debugging:** Understanding exactly how a state was reached.

When clients connect, they often receive an initial "snapshot" of the subscribed data and then continuously receive new events to keep their local view up-to-date.

Visualizing the Real-time Flow

Let's look at a simplified diagram of how a client interaction leads to real-time updates for other clients:



Explanation of the flow: 1. **Client A Calls Reducer:** Client A interacts with the application, which triggers a call to a SpaceTimeDB reducer (e.g., `create_user`). 2. **SpaceTimeDB Executes Reducer:** The SpaceTimeDB instance receives the call and executes the `create_user` reducer. 3. **State Change:** The reducer modifies the SpaceTimeDB database, changing its state. 4. **Generates Events:** SpaceTimeDB internally records this state change as an event (e.g., a new user row was inserted). 5. **Checks Subscriptions:** SpaceTimeDB then checks which connected clients have subscriptions that would be affected by this new event. 6. **Propagates Events:** For every affected client (including Client A, Client B, Client C in this example), SpaceTimeDB pushes the relevant event (the new user data) down their WebSocket connection. All clients see the update in real-time!

Step-by-Step Implementation: Connecting a Web Client

Let's put these concepts into practice. We'll set up a simple HTML page with JavaScript to connect to our SpaceTimeDB instance, subscribe to a table, and react to real-time updates.

Prerequisites: * You have SpaceTimeDB CLI installed and a basic project initialized (from Chapter 2). * You have a SpaceTimeDB module running with at least one table and a reducer. For this example, let's assume we have a `users` table and a `create_user` reducer, similar to what we might have built in Chapter 4.

Example `users` table schema (from Chapter 4, for reference):

```
// In your module's lib.rs or a separate schema file
#[spacetimedb(table)]
pub struct User {
    #[primarykey]
    #[autoinc]
    pub id: u64,
    pub username: String,
    pub created_at: u64,
}

#[spacetimedb(reducer)]
pub fn create_user(ctx: ReducerContext, username: String) -> Result<(),
String> {
    if username.is_empty() {
        return Err("Username cannot be empty".to_string());
    }
    let now = ctx.timestamp;
    User::insert(User { id: 0, username, created_at: now });
    Ok(())
}
```

Make sure your SpaceTimeDB instance is running:

```
spacetime start
spacetime deploy
```

This will typically run on `ws://localhost:3000` by default.

Step 1: Create a Simple HTML File

Let's create an `index.html` file that will host our JavaScript client.

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SpaceTimeDB Real-time Users</title>
    <style>
        body { font-family: sans-serif; margin: 20px; }
        #user-list { border: 1px solid #ccc; padding: 10px; min-height: 100px;
margin-top: 20px; }
        .user-item { margin-bottom: 5px; padding: 5px; background-color: #f9f9f
9; border-radius: 3px; }
        input[type="text"] { padding: 8px; margin-right: 10px; border: 1px soli
d #ddd; }
        button { padding: 8px 15px; background-color: #007bff; color: white; bo
rder: none; border-radius: 3px; cursor: pointer; }
        button:hover { background-color: #0056b3; }
    </style>
</head>
<body>
    <h1>SpaceTimeDB User List</h1>

    <div>
        <input type="text" id="username-input" placeholder="Enter new
username">
        <button id="add-user-button">Add User</button>
    </div>

    <h2>Current Users:</h2>
    <div id="user-list">
        <!-- Users will appear here in real-time -->
        <p>Connecting to SpaceTimeDB...</p>
    </div>

    <!-- Include the SpacetimeDB Client Library -->
    <script src="https://unpkg.com/@clockworklabs/spacetimedb-client@2.0.0/
dist/index.js"></script>
    <script src="app.js"></script>
</body>
</html>
```

Explanation: * We're including the `spacetimedb-client` library directly from unpkg. This is convenient for quick examples. For production, you'd typically install it via npm (`npm install @clockworklabs/spacetimedb-client`) and bundle it with a tool like Webpack or Vite. We're using `v2.0.0` as of 2026-03-14, which is the latest stable major release. * We have a simple input and button to add users, and a `div` to display the list of users. * The `app.js` file is where our SpaceTimeDB client logic will reside.

Step 2: Write the JavaScript Client Logic

Now, create `public/app.js` and add the following code step-by-step.

Part A: Connect to SpaceTimeDB

```
// public/app.js

// 1. Import the SpacetimeDBClient
// If using module bundler: import { SpacetimeDBClient } from '@clockworklabs/
spacetimedb-client';
// Since we're using a script tag, it's globally available as
SpacetimeDBClient.

const SPACETIMEDB_URI = 'ws://localhost:3000'; // Your SpaceTimeDB instance URI
const DB_NAME = 'your_project_name'; // Replace with your actual project name
from `spacetime.toml`

// Create a new client instance
const client = new SpacetimeDBClient(SPACETIMEDB_URI, DB_NAME);

// Get references to DOM elements
const userListDiv = document.getElementById('user-list');
const usernameInput = document.getElementById('username-input');
const addUserButton = document.getElementById('add-user-button');

// Store users locally
let users = {}; // Object to store users by ID for easy lookup and update

// 2. Handle connection status
client.onConnect(() => {
    console.log('Connected to SpaceTimeDB!');
    userListDiv.innerHTML = '<p>Connected. Subscribing to users...</p>';
    // We'll add subscription logic here next
});

client.onDisconnect(() => {
    console.log('Disconnected from SpaceTimeDB.');
    userListDiv.innerHTML = '<p>Disconnected from SpaceTimeDB. Please
refresh.</p>';
});

client.onReconnect(() => {
    console.log('Reconnected to SpaceTimeDB.');
    userListDiv.innerHTML = '<p>Reconnected. Fetching latest data...</p>';
});

client.onError((error) => {
    console.error('SpaceTimeDB Client Error:', error);
    userListDiv.innerHTML = `<p style="color:red;">Error: ${error.message}.
Check console for details.</p>`;
});

// 3. Connect to the database
client.connect();
```

Explanation: * We initialize `SpacetimeDBClient` with the WebSocket URI and your project's database name (found in `spacetime.toml`). Make sure to replace `your_project_name`. * We set up event listeners for `onConnect`, `onDisconnect`, `onReconnect`, and `onError` to provide feedback on the connection status. * Finally, `client.connect()` initiates the WebSocket connection.

Part B: Subscribe to the `users` Table

Now, let's add the subscription logic to `app.js` inside the `onConnect` callback.

```
// ... (previous code) ...

// Store users locally
let users = {}; // Object to store users by ID for easy lookup and update

// Function to render the user list
function renderUserList() {
    userListDiv.innerHTML = ''; // Clear existing list
    const userArray = Object.values(users).sort((a, b) => a.created_at - b.crea
ted_at); // Sort by creation time

    if (userArray.length === 0) {
        userListDiv.innerHTML = '<p>No users yet. Add one!</p>';
        return;
    }

    userArray.forEach(user => {
        const userItem = document.createElement('div');
        userItem.className = 'user-item';
        userItem.textContent = `ID: ${user.id}, Username: ${user.username},
Created: ${new Date(Number(user.created_at)).toLocaleString()}`;
        userListDiv.appendChild(userItem);
    });
}


client.onConnect(() => {
    console.log('Connected to SpaceTimeDB!');
    userListDiv.innerHTML = '<p>Connected. Subscribing to users...</p>';

    // 4. Subscribe to the 'users' table
    // The subscribe method takes the table name as a string.
    client.subscribe(['User']); // 'User' is the name of our table in Rust

    // 5. Listen for table updates
    // These events fire whenever a row is inserted, updated, or deleted in the
'User' table
    client.on('User:onInsert', (userRow, reducerCtx) => {
        console.log('User inserted:', userRow);
        users[userRow.id] = userRow; // Add to local state
        renderUserList();
    });

    client.on('User:onUpdate', (oldUserRow, newUserRow, reducerCtx) => {
        console.log('User updated:', oldUserRow, '->', newUserRow);
        users[newUserRow.id] = newUserRow; // Update local state
        renderUserList();
    });

    client.on('User:onDelete', (userRow, reducerCtx) => {
        console.log('User deleted:', userRow);
        delete users[userRow.id]; // Remove from local state
        renderUserList();
    });

    // 6. Initial data load: After subscription, the client will receive all
existing data.
    // The 'User:onInitialQueryResult' event fires once with all current rows.
    client.on('User:onInitialQueryResult', (initialUsers) => {
        console.log('Initial user query result:', initialUsers);
        users = initialUsers.reduce((acc, user) => {
```

```
            acc[user.id] = user;
            return acc;
        }, {});
        renderUserList();
    });
});

// ... (rest of the code) ...
```

Explanation: * `client.subscribe(['User'])`: This is the crucial line! It tells SpaceTimeDB we want all data from the `User` table. Note that the table name matches the Rust struct name. * `client.on('User:onInsert', ...)`: This event listener fires whenever a new `User` row is inserted into the database by a reducer. * `client.on('User:onUpdate', ...)`: This listener fires when an existing `User` row is modified. * `client.on('User:onDelete', ...)`: This listener fires when a `User` row is removed. * `client.on('User:onInitialQueryResult', ...)`: When you first subscribe, SpaceTimeDB sends all existing data for that table. This event handles that initial batch. We populate our local `users` object and then render. * `renderUserList()`: A helper function to take our `users` object and display it in the HTML.

Part C: Call a Reducer from the Client

Finally, let's add the logic to call our `create_user` reducer when the "Add User" button is clicked.

```
// ... (previous code including onConnect, onInsert, etc.) ...

// 7. Call the create_user reducer when the button is clicked
addUserButton.addEventListener('click', async () => {
    const username = usernameInput.value.trim();
    if (username) {
        try {
            console.log(`Calling create_user reducer with username: ${username}
`);
            // The callReducer method takes the reducer name and its arguments
            await client.callReducer('create_user', username);
            usernameInput.value = ''; // Clear input after successful call
        } catch (error) {
            console.error('Error calling create_user reducer:', error);
            alert(`Failed to add user: ${error.message}`);
        }
    } else {
        alert('Please enter a username!');
    }
});

// 8. Connect to the database
client.connect();
```

Explanation: * We add an event listener to the `addUserButton`. * Inside the listener, `client.callReducer('create_user', username)` sends a request to SpaceTimeDB to execute our `create_user` reducer with the provided `username`. * Since the reducer modifies the `User` table, SpaceTimeDB will automatically detect this change and push an `onInsert` event to all subscribed clients, including our own. This will trigger our `onInsert` listener and update the UI in real-time!

Step 3: Test Your Real-time Application

1. **Ensure SpaceTimeDB is running:** `bash spacetime start spacetime deploy` Verify it's running on `localhost:3000`.
2. **Open `public/index.html` in your browser.** You can simply drag and drop the file into your browser, or serve it with a simple local web server (e.g., Python's `http.server` or `live-server` npm package). `bash # From your project root, assuming public/index.html and public/app.js cd public python3 -m http.server 8000 # Then open http://localhost:8000 in your browser`
3. **Observe:**
    - You should see "Connected to SpaceTimeDB..." and then "No users yet. Add one!"
    - Open your browser's developer console (F12) to see the `console.log` messages.
    - Type a username into the input field and click "Add User".
    - The user should appear instantly in the "Current Users" list!
    - **Crucially, open a second browser tab or window to `http://localhost:8000`. When you add a user in one tab, it should appear instantly in the other tab as well, without refreshing!** This is the real-time magic in action.

Mini-Challenge: Enhance User Display

You've seen the basic real-time update. Now, let's make it a bit more interactive.

Challenge: Modify the `renderUserList` function and the `user-item` styling so that: 1. Each user item includes a "Delete" button. 2. Clicking the "Delete" button calls a new SpaceTimeDB reducer named `delete_user` that takes a `user_id` and removes the corresponding user from the `User` table. 3. Observe that

deleting a user in one browser tab instantly removes it from all other connected tabs.

Hint: * You'll need to define a new `delete_user` reducer in your Rust module (`lib.rs`). It should take `user_id: u64` and use `User::delete(user_id)` to remove the row. Remember to handle potential errors (e.g., user not found). * In `renderUserList`, create the button element, attach an `onclick` event listener that calls `client.callReducer('delete_user', userId)`, and append it to the `userItem`. * Remember to handle the `User:onDelete` event in your JavaScript client to update the `users` object and re-render.

Common Pitfalls & Troubleshooting

1. **Incorrect SpaceTimeDB URI or DB Name:**

- **Symptom:** Client shows "Disconnected" or "Error: WebSocket connection failed."
- **Fix:** Double-check `SPACETIMEDB_URI` (default `ws://localhost:3000`) and `DB_NAME` in `app.js`. The `DB_NAME` must match the `name` field in your `spacetime.toml` file. 2. **Table Name Mismatch:**
- **Symptom:** Client connects but doesn't receive any data or updates for subscribed tables.
- **Fix:** Ensure the table name passed to `client.subscribe(['TableName'])` exactly matches the Rust struct name defined with `#[spacetimedb(table)]` (e.g., `User` not `users`). Case sensitivity matters! 3. **Reducer Logic Errors:**
- **Symptom:** Reducer calls complete without error, but no data appears, or incorrect data appears.
- **Fix:** Check your Rust reducer logic (`lib.rs`). Does it correctly `insert`, `update`, or `delete` the intended data? Use `console.log` on the client side and `spacetime log` on the server side to trace reducer execution and database changes. 4. **No `client.connect()`:**
- **Symptom:** Nothing happens, no connection attempts.
- **Fix:** Ensure `client.connect()` is called at the end of your `app.js` file. 5. **Running Client Before SpaceTimeDB:**
- **Symptom:** WebSocket connection errors.

- **Fix:** Always ensure your SpaceTimeDB instance is running (`spacetime start` and `spacetime deploy`) before opening your client application.

Summary

Congratulations! You've just built your first truly real-time application using SpaceTimeDB. You've seen how SpaceTimeDB seamlessly integrates database management, backend logic, and real-time synchronization into a single, powerful platform.

Here are the key takeaways from this chapter: * SpaceTimeDB uses **WebSockets** for persistent, bidirectional client-server communication. * **Subscriptions** are the declarative way for clients to express interest in specific data, enabling SpaceTimeDB to push relevant updates. * SpaceTimeDB generates **events** for every database change, which are then propagated to subscribed clients. * The **deterministic event-sourced architecture** ensures consistency across all connected clients. * The `@clockworklabs/spacetimedb-client` library provides a straightforward API for connecting, subscribing, and calling reducers from web applications.

You're now equipped to start building dynamic, collaborative, and highly responsive applications that leverage SpaceTimeDB's real-time capabilities.

What's Next? In the next chapter, we'll dive deeper into more advanced subscription patterns, including how to filter and query data efficiently on the client side, and explore how to structure more complex multiplayer or collaborative application patterns.

References

- SpacetimeDB Official Documentation
- SpacetimeDB GitHub Repository
- SpacetimeDB Client Library on unpkg
- MDN Web Docs - WebSockets

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 15

Chapter 7: Building Collaborative Features: Patterns for Shared State

Chapter 7: Building Collaborative Features: Patterns for Shared State

Welcome back, intrepid developer! In our journey through SpaceTimeDB, we've covered the basics of setting up your database, defining schemas, and even writing server-side logic with reducers. But where SpaceTimeDB truly shines is in its ability to power real-time, collaborative applications. This is where the magic of shared state and instant synchronization comes alive!

In this chapter, we're going to dive deep into building collaborative features. We'll explore the patterns and techniques that allow multiple users to interact with the same data simultaneously, seeing updates happen in real-time across all connected clients. Think multiplayer games, shared whiteboards, collaborative document editors, or live dashboards – SpaceTimeDB makes these complex scenarios surprisingly approachable. Get ready to build applications that feel alive and responsive!

To make the most of this chapter, you should be comfortable with: * SpaceTimeDB installation and basic project setup (Chapter 2) * Schema definition with tables and fields (Chapter 3) * Writing and deploying SpaceTimeDB reducers (Chapter 5) * Basic client-side interaction with SpaceTimeDB, including subscriptions (Chapter 6)

The Heart of Collaboration: Shared State and Real-time Sync

At its core, a collaborative application is all about shared state. It's about ensuring that everyone involved sees and interacts with the same, consistent version of reality. If one user draws a line, every other user should see that line appear instantly. If a player moves their character, all other players should witness that movement without delay.

Traditional backend architectures often struggle with this. You might have to poll the server constantly, manage complex WebSocket connections, or build

elaborate pub/sub systems. SpaceTimeDB, however, is designed from the ground up to handle this automatically:

1. **Centralized, Deterministic State:** Your entire application state lives in SpaceTimeDB tables. All changes to this state happen through deterministic reducers. This means that for a given starting state and a sequence of reducer calls, the final state will always be the same, regardless of when or where the reducer was called. This determinism is crucial for consistency.
2. **Event-Driven Updates:** Whenever a reducer successfully modifies the database, SpaceTimeDB treats this as an "event." It then efficiently propagates these changes to all subscribed clients. You don't need to manually push updates; SpaceTimeDB handles the "when" and "how."
3. **Client-Side Reactivity:** The client SDKs (like TypeScript/JavaScript) automatically update their local views of the subscribed tables. This reactive nature means your frontend code can simply render the current state, and it will automatically reflect any changes from other users.

Let's visualize this flow:

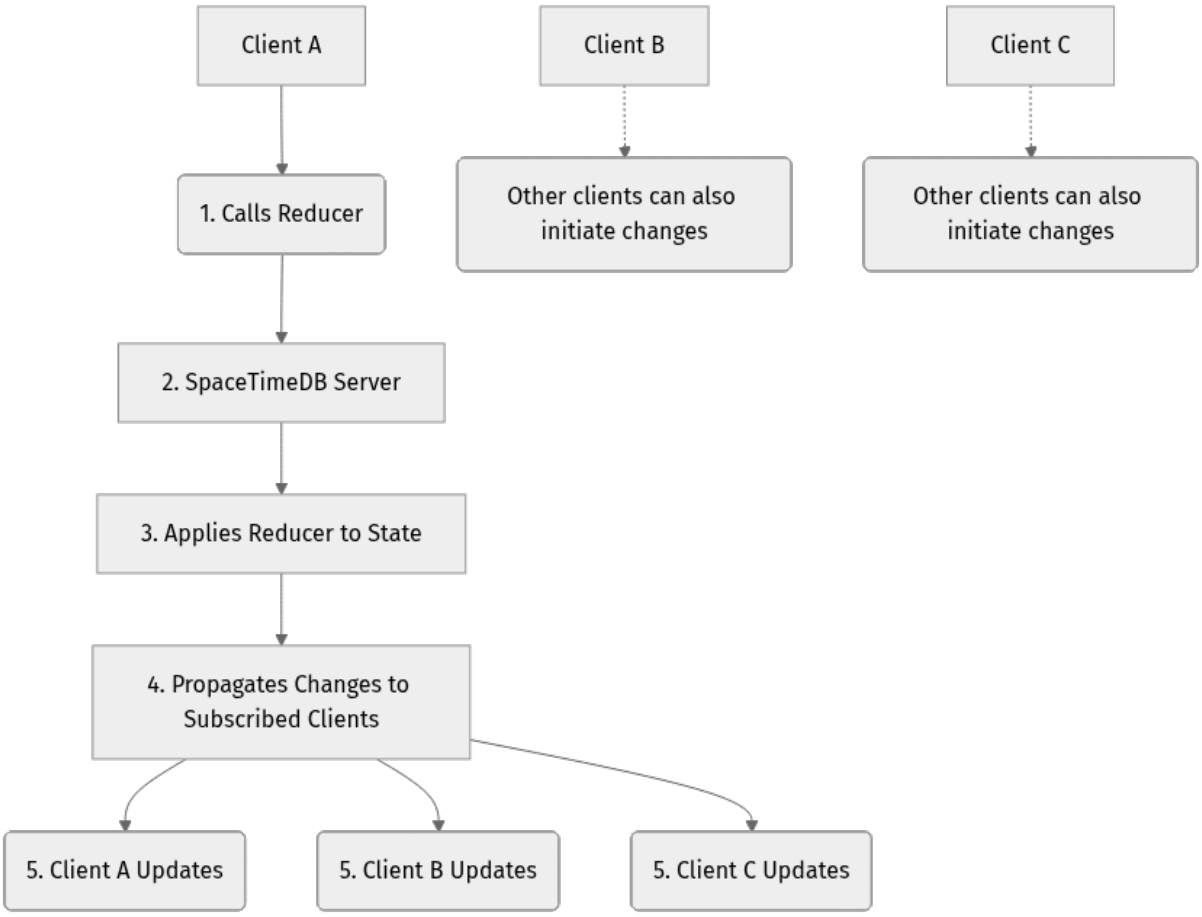


Figure 7.1: SpaceTimeDB's Shared State Synchronization Flow

As you can see, when Client A calls a reducer, SpaceTimeDB processes it, updates its internal state, and then broadcasts those changes to all clients that are subscribed to the affected tables, including Client A itself. This ensures everyone is always in sync.

Optimistic vs. Pessimistic Concurrency

When multiple users try to modify the same data, concurrency becomes a concern. * **Pessimistic concurrency** involves locking data before modifying it, preventing others from accessing it until the lock is released. This ensures consistency but can hurt responsiveness and scalability. * **Optimistic concurrency** assumes conflicts are rare. Users modify data, and conflicts are detected and resolved after the fact. This is generally more scalable and responsive.

SpaceTimeDB leans towards an optimistic model, but with a twist. Its deterministic reducers guarantee that operations are applied in a strict, consistent order on the server. If two clients call the same reducer "simultaneously," SpaceTimeDB will process them one after another. Clients will then receive the updates in the order they were processed, ensuring everyone ends up with the same final state. Conflicts are implicitly resolved by the server's single-threaded, deterministic execution model for reducers.

Common Collaborative Patterns

Let's look at some patterns you'll encounter when building collaborative features:

1. **Presence:** Knowing who is online and what they're doing. This is fundamental for many collaborative apps (e.g., "3 users are viewing this document").
2. **Shared Cursors/Selections:** In a text editor, seeing where others are typing or what they have selected.
3. **Shared Canvas/Whiteboard:** Multiple users drawing on the same digital canvas. This is a fantastic way to demonstrate real-time interaction.
4. **Multiplayer Game State:** Synchronizing player positions, scores, game events, and item interactions.

We'll focus on building a simple **Shared Canvas/Whiteboard** example to illustrate these concepts.

Step-by-Step Implementation: A Collaborative Whiteboard

Let's build a minimalist collaborative whiteboard where multiple users can draw points, and everyone sees them appear instantly.

Step 1: Define the Schema for Drawing Points

First, we need to define how we'll store individual points drawn on our whiteboard.

Open your `spacetime-module/src/lib.rs` file (or create one if you're starting a new module) and add the following table definition.

```
// spacetime-module/src/lib.rs

use spacetimedb::{spacetimedb, Table, Reducer, Identity, Timestamp};

// Our Point table will store each individual dot drawn on the canvas.
#[spacetimedb(table)]
pub struct Point {
    #[spacetimedb(primarykey)]
    pub id: u64, // A unique ID for each point
    pub x: f32,  // X coordinate
    pub y: f32,  // Y coordinate
    pub color: String, // Color of the point (e.g., "#FF0000")
    pub user_id: Identity, // The identity of the user who drew this point
    pub timestamp: Timestamp, // When the point was drawn
}

// We'll add our reducer here next!
```

Explanation:

- **`Point` struct:** Represents a single point drawn on the canvas.
- **`id: u64`:** A unique identifier for each point. We'll generate this on the server to ensure uniqueness.
- **`x: f32`, `y: f32`:** Floating-point coordinates for precision.
- **`color: String`:** Stores the color, perhaps as a hex string (e.g., `#RRGGBB`).
- **`user_id: Identity`:** This is crucial for collaboration! SpaceTimeDB's `Identity` type automatically tracks the unique identifier of the client calling the reducer. This allows us to attribute points to specific users.
- **`timestamp: Timestamp`:** Records when the point was created. `Timestamp` is another built-in SpaceTimeDB type.

Step 2: Create a Reducer to Add Points

Now, let's create a reducer that allows clients to add new points to our `Point` table.

Add this reducer to your `spacetime-module/src/lib.rs` file, right after the `Point` struct.

```
// spacetime-module/src/lib.rs
// ... (Point struct definition) ...

#[spacetimedb(reducer)]
pub fn add_point(ctx: ReducerContext, x: f32, y: f32, color: String) {
    // Generate a unique ID for the new point.
    // In a real app, you might use a more robust ID generation strategy
    // or SpaceTimeDB's built-in sequence generator if available.
    // For now, we'll use a simple timestamp-based ID.
    let id = ctx.timestamp.as_micros();

    // Insert the new point into the Point table.
    Point::insert(Point {
        id,
        x,
        y,
        color,
        user_id: ctx.sender, // The user who called this reducer
        timestamp: ctx.timestamp,
    }).expect("Failed to insert point");

    // No explicit return needed; the side effect of inserting the row
    // is what matters and will be propagated.
}
```

Explanation:

- **`#[spacetimedb(reducer)]`:** Marks this function as a SpaceTimeDB reducer.
- **`ctx: ReducerContext`:** This special parameter provides useful context about the reducer call, including:
    - `ctx.sender`: The `Identity` of the client that invoked this reducer. Perfect for attributing actions!
    - `ctx.timestamp`: The server's current timestamp when the reducer is executed. Great for unique IDs and ordering.
- **`x: f32, y: f32, color: String`:** These are the parameters passed from the client, representing the details of the point to be drawn.
- **`id = ctx.timestamp.as_micros()`:** We're using the reducer's execution timestamp (in microseconds) as a simple unique ID. While usually effective, for highly concurrent systems, a dedicated counter or UUID generation might be considered if the timestamp resolution isn't sufficient for uniqueness.
- **`Point::insert(...)`:** This is how we add a new row to the `Point` table. We construct a `Point` instance using the provided arguments and the `ctx.sender` and `ctx.timestamp`.
- **`expect("Failed to insert point")`:** Basic error handling. In production, you'd want more graceful error management.

Step 3: Deploy the Module

Save your `lib.rs` file. Now, navigate to your `spacetime-module` directory in your terminal and deploy your changes:

```
spacetime deploy
```

This will compile your Rust module and push it to your SpaceTimeDB instance. If you're running a local `spacetime-dev` instance, it will be updated instantly.

Step 4: Client-Side: Connecting and Drawing

Now for the fun part! Let's build a simple web client using TypeScript to connect, draw, and see updates.

Setup your client project: If you don't have a client project set up, create a simple `index.html` and `script.ts` (or `script.js`) file. You'll need to compile TypeScript if you use `.ts`.

`client/index.html` (minimal HTML):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SpaceTimeDB Whiteboard</title>
    <style>
        body { font-family: sans-serif; display: flex; flex-direction: column;
align-items: center; margin: 20px; }
        canvas { border: 1px solid #ccc; background-color: #f9f9f9; cursor: cro
sshair; }
        .controls { margin-top: 10px; }
        .color-picker { width: 40px; height: 40px; border: none; cursor: pointe
r; }
        .user-info { margin-bottom: 15px; font-size: 0.9em; color: #555; }
    </style>
</head>
<body>
    <h1>Collaborative Whiteboard</h1>
    <div class="user-info" id="user-info">Connecting...</div>
    <div class="controls">
        Choose Color: <input type="color" id="colorPicker" class="color-
picker" value="#FF0000">
    </div>
    <canvas id="whiteboardCanvas" width="800" height="600"></canvas>

    <script src="./script.js"></script> <!-- Ensure this matches your compiled
JS file -->
</body>
</html>
```

`client/script.ts` (Client-side logic):

```
// client/script.ts

import { SpacetimeDBClient, Identity } from "@clockworklabs/spacetimedb-sdk";
import { Point } from "./types"; // We'll generate this soon!
import { add_point } from "./modules/spacetime"; // And this!

// --- Configuration ---
const SPACETIMEDB_URI = "ws://localhost:3000"; // Your SpaceTimeDB instance URI
const DB_NAME = "my_whiteboard_db"; // Replace with your actual DB name if
different

// --- DOM Elements ---
const canvas = document.getElementById("whiteboardCanvas") as
HTMLCanvasElement;
const ctx = canvas.getContext("2d")!;
const colorPicker = document.getElementById("colorPicker") as HTMLInputElement;
const userInfo = document.getElementById("user-info") as HTMLDivElement;

let currentColor: string = colorPicker.value;
let currentIdentity: Identity | null = null; // Store the current user's
identity

// --- SpaceTimeDB Client Setup ---
const client = new SpacetimeDBClient(SPACETIMEDB_URI);

client.onConnect(() => {
    console.log("Connected to SpaceTimeDB!");
    client.enter(DB_NAME);
});

client.onDisconnect(() => {
    console.log("Disconnected from SpaceTimeDB.");
});

client.onInitialSync(() => {
    currentIdentity = client.identity;
    console.log("Initial sync complete. My identity:", currentIdentity?.to_stri
ng());
    userInfo.textContent = `Connected as: ${currentIdentity?.to_string().substr
ing(0, 8)}...`;

    // Subscribe to the 'Point' table to receive all drawing updates
    client.subscribe([`Point`]);
});

// --- Client-side Type Definitions (for generated types) ---
// For now, we manually define Point. Later, we'll use generated types.
// Create a file: `client/types.ts`
/*
export interface Point {
    id: bigint; // u64 in Rust becomes bigint in TypeScript
    x: number;
    y: number;
    color: string;
    user_id: Identity;
    timestamp: bigint; // Timestamp in Rust becomes bigint
}
*/
// You'll need to generate these types from your SpaceTimeDB module.
// Run `spacetime client-sdk generate --lang ts --out-dir client`
// This will create `client/types.ts` and `client/modules/spacetime.ts`
```

```
// --- Drawing Logic ---
let isDrawing = false;

canvas.addEventListener("mousedown", (e) => {
    isDrawing = true;
    draw(e); // Draw the first point immediately
});

canvas.addEventListener("mousemove", (e) => {
    if (!isDrawing) return;
    draw(e);
});

canvas.addEventListener("mouseup", () => {
    isDrawing = false;
    ctx.beginPath(); // Reset path for next drawing stroke
});

canvas.addEventListener("mouseout", () => {
    isDrawing = false;
    ctx.beginPath();
});

colorPicker.addEventListener("change", (e) => {
    currentColor = (e.target as HTMLInputElement).value;
});

function draw(e: MouseEvent) {
    if (!currentIdentity) {
        console.warn("Not connected yet, cannot draw.");
        return;
    }

    const rect = canvas.getBoundingClientRect();
    const x = e.clientX - rect.left;
    const y = e.clientY - rect.top;

    // Call the SpaceTimeDB reducer!
    add_point(x, y, currentColor);

    // Locally draw the point immediately for responsiveness
    // (This point will be redrawn when the server sends the update,
    // but drawing it locally prevents perceived lag)
    drawSinglePoint(x, y, currentColor);
}

function drawSinglePoint(x: number, y: number, color: string) {
    ctx.lineTo(x, y);
    ctx.stroke();
    ctx.beginPath();
    ctx.moveTo(x, y);

    ctx.strokeStyle = color; // Set stroke color
    ctx.lineWidth = 2; // Set line width
    ctx.lineCap = "round"; // Round line ends
}

// --- SpaceTimeDB Table Listeners ---

// This function clears the canvas and redraws all points.
// We'll call this whenever the 'Point' table changes.
```

```
function redrawAllPoints() {
    ctx.clearRect(0, 0, canvas.width, canvas.height); // Clear entire canvas
    ctx.beginPath(); // Reset path

    // Get all points from the client's local cache of the 'Point' table
    const allPoints = client.get_all("Point") as Point[];

    // Sort points by timestamp to ensure drawing order is consistent
    allPoints.sort((a, b) => Number(a.timestamp - b.timestamp));

    allPoints.forEach((point, index) => {
        ctx.strokeStyle = point.color;
        ctx.lineWidth = 2;
        ctx.lineCap = "round";

        // For simplicity, we're drawing points as individual strokes.
        // A more advanced drawing app would group points into strokes.
        if (index === 0 || point.id !== allPoints[index - 1].id) { // Simple
check for new stroke
            ctx.beginPath();
            ctx.moveTo(point.x, point.y);
        } else {
            ctx.lineTo(point.x, point.y);
            ctx.stroke();
            ctx.beginPath();
            ctx.moveTo(point.x, point.y);
        }
    });
}

// Register a listener for changes to the 'Point' table
client.on("Point:insert", redrawAllPoints);
client.on("Point:update", redrawAllPoints); // Not strictly needed for this
example, but good practice
client.on("Point:delete", redrawAllPoints); // If we add a 'clear' reducer
later

// Initial draw when client connects and syncs
client.onInitialSync(redrawAllPoints);

// Connect to SpaceTimeDB
client.connect();
```

Explanation of Client-Side Code:

1. **Imports:** We import `SpacetimeDBClient` and `Identity` from the SDK. We also anticipate importing `Point` and `add_point` from generated files.
2. **Configuration:** Set your SpaceTimeDB URI and database name.
3. **DOM Elements:** Get references to your canvas and color picker.
4. **`SpacetimeDBClient` Setup:**
    - `client.onConnect`, `client.onDisconnect`: Basic connection logging.

    - `client.onInitialSync`: This is crucial! Once the client has received the initial snapshot of the database, we can:
        - Get the client's `Identity` (SpaceTimeDB assigns one automatically if not authenticated).

- **`client.subscribe([Point])`:** Tell SpaceTimeDB we want to receive all updates for the `Point` table. This is how real-time synchronization happens! 5. **Drawing Logic:**
    - `mousedown`, `mousemove`, `mouseup` event listeners handle user interaction on the canvas.

- **`add_point(x, y, currentColor)`:** When a user draws, we directly call our SpaceTimeDB reducer from the client! This sends the drawing action to the server.
- **`drawSinglePoint`:** This function draws a point directly on the local canvas. We do this optimistically for immediate visual feedback. The "true" state will come from the server, but this makes the app feel snappier. 6. **`redrawAllPoints()`:** This is the core of our collaborative rendering.
    - It clears the entire canvas.
    - It fetches all points currently in the client's local cache of the `Point` table using `client.get_all("Point")`.
    - It then iterates through these points and redraws them. Sorting by `timestamp` ensures a consistent drawing order across clients. 7. **`client.on("Point:insert", redrawAllPoints)`:** This is the magic! Whenever a new `Point` is inserted (by any client, including our own after the reducer call), this listener fires, and we redraw the entire canvas. This ensures all clients see the same, up-to-date drawing.

Step 5: Generate Client-Side Types and Reducer Stubs

Before running the client, you need to generate the TypeScript types for your tables and the stub functions for your reducers. This makes client-side development much easier and type-safe.

Navigate to your project's root (or wherever you want the `client` folder to be) and run:

```
spacetime client-sdk generate --lang ts --out-dir client
```

This command will create: * `client/types.ts`: Contains TypeScript interfaces matching your Rust `#[spacetimedb(table)]` structs. * `client/modules/`

`spacetime.ts`: Contains TypeScript functions that wrap your Rust `#[spacetimedb(reducer)]` functions, allowing you to call them directly from your client-side code.

Verify that `client/types.ts` contains an interface for `Point` and `client/modules/spacetime.ts` contains an `add_point` function. If you're using a bundler like Webpack or Parcel, you might need to adjust paths slightly. For a simple setup, ensure your `script.ts` can import from these generated files.

Step 6: Run Your Client

You'll need a way to serve your `index.html` and `script.js` (compiled from `script.ts`). A simple HTTP server will do:

```
# If you have Node.js and npm installed:
npm install -g http-server
cd client
http-server . -p 8080
```

Now, open your browser to `http://localhost:8080`. Open a second browser tab (or even a different browser!) to the same URL. Draw in one window, and watch it appear in the other! You've just built a real-time collaborative application with SpaceTimeDB!

Mini-Challenge: Clear My Drawings

Let's enhance our whiteboard. It's great to draw, but sometimes you want a fresh start, or maybe just want to erase your own contributions.

Challenge: Add a new reducer called `clear_my_points` that, when called, deletes all `Point` entries that belong to the `Identity` of the caller. Then, add a button to your HTML and connect it to this new reducer in your client-side TypeScript.

Hint: * You'll need to add a `#[spacetimedb(reducer)]` function to `lib.rs`. * Inside the reducer, you can use `Point::filter(|point| point.user_id == ctx.sender).delete_all();` * Remember to `spacetime deploy` after modifying `lib.rs`. * After deployment, regenerate your client SDK types (`spacetime client-sdk generate ...`) so the `clear_my_points` function is available in your client. * In your client `script.ts`, import the new reducer and add an event listener to a button.

What do you observe when you clear your points? Do other users see your points disappear? Why or why not?

Common Pitfalls & Troubleshooting

1. **Forgetting to `spacetime deploy`:** Any changes to your Rust module (`lib.rs`) require a `spacetime deploy` to take effect on the server. Always remember this step!
2. **Not regenerating client SDK:** If you add or change tables/reducers in your Rust module, your client-side TypeScript definitions will be out of date. Always run `spacetime client-sdk generate --lang ts --out-dir client` after a `spacetime deploy` if your client code relies on those changes.
3. **Incorrect Subscriptions:** If your client isn't receiving updates, double-check that you've correctly called `client.subscribe([TableName])` for all relevant tables. No subscription, no real-time updates!
4. **Client-side Redrawing Logic:** For complex collaborative apps, simply `redrawAllPoints()` on every update might be inefficient. For simple cases like our whiteboard, it's fine. For a game, you might update specific entities. Consider if you need to optimize rendering by only redrawing changed elements, though SpaceTimeDB's reactivity often makes full redraws surprisingly performant for many applications.
5. **Race Conditions in Client-Side Logic:** While SpaceTimeDB's reducers are deterministic, your client-side code might still have race conditions if it relies on local state that hasn't been confirmed by the server. Always strive to make your client-side rendering a pure function of the SpaceTimeDB state. If you draw optimistically (like we did), ensure the server's update eventually reconciles that state.

Summary

Phew! You've just built a truly collaborative application. In this chapter, we covered:

- The fundamental principles of shared state and real-time synchronization in SpaceTimeDB.
- How SpaceTimeDB's deterministic reducers and event-driven propagation simplify building collaborative features.
- Common collaborative patterns like presence and shared canvases.
- A hands-on example of building a real-time collaborative whiteboard using SpaceTimeDB tables and reducers.
- The importance of `client.subscribe()` for receiving real-time updates.
- Key steps for client-side development, including generating SDK types.

- Practical tips and troubleshooting for common issues.

You now have a solid understanding of how to leverage SpaceTimeDB for real-time, multi-user applications. This opens up a world of possibilities, from multiplayer games to live data dashboards and interactive productivity tools.

What's Next?

In the next chapter, we'll delve into more advanced topics like managing user authentication and authorization, ensuring that only the right users can perform specific actions in your collaborative applications.

References

- SpacetimeDB Official Documentation
- SpacetimeDB GitHub Repository
- SpacetimeDB Client SDK for TypeScript/JavaScript

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 16

Chapter 8: Integrating with Frontends: Web Clients and Game Engines

Introduction

Welcome to Chapter 8! So far, we've explored the fascinating world of SpaceTimeDB, understanding its core concepts, how to define schemas, and how to implement server-side logic using reducers. We've built the "brain" of our real-time applications, where data lives and logic executes deterministically.

But what's a powerful backend without a beautiful and interactive frontend? This chapter is all about bridging that gap. We'll dive deep into how your client applications—whether they're web apps built with JavaScript/TypeScript or games developed with engines like Unity using C#—connect to SpaceTimeDB, subscribe to real-time data updates, and invoke your server-side reducers. By the end of this chapter, you'll be able to bring your SpaceTimeDB-powered ideas to life with dynamic, real-time user interfaces.

This chapter assumes you have a basic understanding of frontend development (HTML, CSS, JavaScript for web, or C# for Unity) and have a SpaceTimeDB instance running with some basic schema and reducers from previous chapters. Let's make our applications truly interactive!

Core Concepts: Connecting Your Frontend to the Real-time Backend

Integrating a frontend with SpaceTimeDB is a seamless experience designed for real-time interaction. It leverages WebSockets for persistent connections and dedicated client SDKs to simplify the process.

The Role of Client SDKs

SpaceTimeDB provides client-side Software Development Kits (SDKs) for various languages and platforms. These SDKs are your primary interface for interacting with your SpaceTimeDB instance. They handle:

- **WebSocket Connection Management:** Establishing and maintaining a persistent WebSocket connection to your SpaceTimeDB server.

- **Serialization/Deserialization:** Converting your client-side data into a format SpaceTimeDB understands, and vice-versa.
- **Real-time Event Handling:** Listening for and processing database changes pushed from the server.
- **Reducer Invocation:** Providing a convenient way to call your server-side reducers from the client.
- **Local Cache Management (Optional):** Some SDKs might offer mechanisms for maintaining a local, consistent view of subscribed data, reducing the need for you to build this from scratch.

For web applications, the primary SDK is the JavaScript/TypeScript SDK. For game engines like Unity, a C# SDK is available.

Real-time Synchronization: How Data Flows

The magic of SpaceTimeDB lies in its real-time synchronization. When a client connects and subscribes to a table, here's the general flow:

1. **Initial Sync:** Upon successful connection and subscription, the client receives the current state of all subscribed tables. This is often referred to as the "initial sync."
2. **Event Stream:** After the initial sync, the client continuously receives a stream of "update" events whenever data in the subscribed tables changes on the server. These updates are granular, often indicating which rows were inserted, updated, or deleted.
3. **Client-Side Update:** Your frontend application listens for these events and updates its UI or internal state accordingly, reflecting the latest changes in real-time.

This push-based model, powered by WebSockets, eliminates the need for manual polling, making your applications highly responsive and efficient.

Calling Reducers from the Client

Clients don't directly modify the database. Instead, they invoke **reducers** defined on the SpaceTimeDB server. This is a critical security and consistency feature:

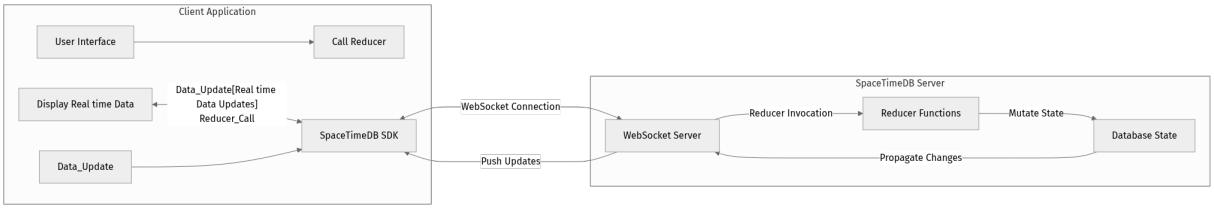
- **Centralized Logic:** All state-modifying logic resides on the server, ensuring all clients operate under the same rules.
- **Deterministic Execution:** Reducers execute deterministically, guaranteeing consistency across all replicas and clients.

- **Security:** Access control and validation logic can be enforced within reducers, preventing unauthorized or invalid operations.

When a client calls a reducer, the SDK sends the reducer's name and its arguments to the SpaceTimeDB server. The server executes the reducer, and if successful, propagates the resulting database changes back to all subscribed clients.

Architecture Diagram: Client-Server Interaction

Let's visualize this interaction:



- **Client Application:** This is your web browser or game client. It interacts with the SDK.
- **SpaceTimeDB SDK:** The client-side library that manages the WebSocket connection and handles communication.
- **WebSocket Connection:** The persistent, bidirectional link between the client and the server.
- **Reducer Invocation:** When the client wants to perform an action (like sending a message), it calls a reducer via the SDK.
- **Reducers:** Server-side functions that contain your business logic and modify the database.
- **Database State:** The core data managed by SpaceTimeDB.
- **Propagate Changes:** After a reducer modifies the database, SpaceTimeDB intelligently identifies the changes.
- **Push Updates:** These changes are then pushed in real-time over the WebSocket connection to all subscribed clients.
- **Real-time Data Updates:** The client SDK receives these updates, allowing the UI to react instantly.

Client-Side State Management

While SpaceTimeDB handles the source of truth and real-time propagation, your frontend still needs its own strategy for managing its local state. This could be:

- **Direct DOM Manipulation (Vanilla JS):** Simple for small examples.

- **Framework-Specific State (React Hooks, Vuex, Angular Services):** Integrating SpaceTimeDB updates into your chosen framework's reactive system.
- **Game Engine State (Unity MonoBehaviours):** Updating game objects and components based on incoming data.

The key is to connect SpaceTimeDB's update events to your frontend's state management system so that your UI or game world reflects the latest shared state.

Step-by-Step Implementation: A Simple Web Chat Client

Let's build a basic web chat application to demonstrate how to integrate SpaceTimeDB. We'll assume you have a SpaceTimeDB project set up from previous chapters with a `Message` table and a `create_message` reducer.

Prerequisites:

1. A running SpaceTimeDB instance (e.g., on `localhost:3000`).
2. A SpaceTimeDB module with a `Message` table and a `create_message` reducer.

- **`lib.stdb` (Schema):** `rust // Define a Message table #[derive(SpacetimeDBType, Clone, PartialEq, Eq, Debug)] pub struct Message { #[primarykey] #[autoinc] pub id: u64, pub sender: String, pub text: String, pub timestamp: u64, // Unix timestamp }`
- **`reducers.stdb` (Reducer):** ```rust use super::message::Message;

```
#[reducer]
pub fn create_message(sender: String, text: String) {
    let timestamp = spacetime::now(); // Get current timestamp from
SpaceTimeDB
    Message::insert(Message {
        id: 0, // autoinc will assign a real ID
        sender,
        text,
        timestamp,
    }).unwrap();
}
```

- Ensure your SpaceTimeDB module is compiled and running (e.g., `spacetime db dev`).

Step 1: Set up a Basic Web Project

Create a new folder named `chat-frontend`. Inside it, create `index.html` and `script.js`.

`chat-frontend/index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>SpaceTimeDB Chat</title>
    <style>
        body { font-family: sans-serif; margin: 20px; background-color:
#f4f4f4; }
        #chat-container {
            max-width: 600px;
            margin: 0 auto;
            background-color: #fff;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        }
        #messages {
            border: 1px solid #ddd;
            height: 300px;
            overflow-y: scroll;
            padding: 10px;
            margin-bottom: 15px;
            border-radius: 4px;
            background-color: #e9e9e9;
        }
        .message {
            margin-bottom: 8px;
            line-height: 1.4;
        }
        .message strong { color: #333; }
        .message span { color: #666; font-size: 0.9em; }
        #message-form { display: flex; gap: 10px; }
        #username-input, #message-input {
            flex-grow: 1;
            padding: 10px;
            border: 1px solid #ccc;
            border-radius: 4px;
        }
        #send-button {
            padding: 10px 15px;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            transition: background-color 0.2s;
        }
        #send-button:hover { background-color: #0056b3; }
        #connection-status {
            text-align: center;
            margin-bottom: 10px;
            font-size: 0.9em;
            color: #555;
        }
        .status-connected { color: green; }
        .status-disconnected { color: red; }
    </style>
</head>
<body>
    <div id="chat-container">
```

```
        <h1>SpaceTimeDB Real-time Chat</h1>
        <div id="connection-status">Connecting...</div>
        <div id="messages">
            <!-- Messages will be rendered here -->
        </div>
        <form id="message-form">
            <input type="text" id="username-input" placeholder="Your Name" requ
ired>
            <input type="text" id="message-input" placeholder="Type a
message..." required>
            <button type="submit" id="send-button">Send</button>
        </form>
    </div>

    <script type="module" src="script.js"></script>
</body>
</html>
```

This is a standard HTML file with some basic styling and elements for our chat interface: a message display area, an input for username, an input for the message, and a send button. The `<script type="module" src="script.js"></script>` line is important as it allows us to use ES module syntax in `script.js`.

Step 2: Initialize Node.js Project and Install SpaceTimeDB SDK

Open your terminal in the `chat-frontend` directory.

1. **Initialize Node.js project:** `bash npm init -y` This creates a `package.json` file.
2. **Install SpaceTimeDB SDK:** `bash npm install @clockworklabs/spacetimedb-sdk@2.0.1 --save-exact` We're explicitly installing `v2.0.1` as of 2026-03-14, which is a recent stable release for the SDK. The `--save-exact` flag ensures this specific version is recorded.

   Why this package? The `@clockworklabs/spacetimedb-sdk` is the official JavaScript/TypeScript client library for SpaceTimeDB, abstracting away the WebSocket communication and providing convenient methods for interacting with your module.

Step 3: Connect to SpaceTimeDB and Subscribe to Messages

Now, let's write the JavaScript code to connect and receive messages.

`chat-frontend/script.js`:

Start by importing the SDK and setting up basic DOM references.

```
// chat-frontend/script.js

import { SpacetimeDBClient, Identity } from '@clockworklabs/spacetimedb-sdk';

// --- DOM Element References ---
const messagesDiv = document.getElementById('messages');
const messageForm = document.getElementById('message-form');
const usernameInput = document.getElementById('username-input');
const messageInput = document.getElementById('message-input');
const connectionStatusDiv = document.getElementById('connection-status');

// --- Configuration ---
const SPACETIMEDB_HOST = 'localhost';
const SPACETIMEDB_PORT = 3000;
const SPACETIMEDB_DB_NAME = 'chat_module'; // Replace with your module's name

// --- Utility Functions ---
function appendMessage(sender, text, timestamp) {
    const messageElement = document.createElement('div');
    messageElement.classList.add('message');
    const date = new Date(Number(timestamp) * 1000); // SpaceTimeDB timestamp
is in seconds
    messageElement.innerHTML = `<strong>${sender}</strong>: ${text} <span>(${da
te.toLocaleTimeString()})</span>`;
    messagesDiv.appendChild(messageElement);
    messagesDiv.scrollTop = messagesDiv.scrollHeight; // Scroll to bottom
}

function updateConnectionStatus(isConnected) {
    if (isConnected) {
        connectionStatusDiv.textContent = 'Connected to SpaceTimeDB';
        connectionStatusDiv.className = 'status-connected';
    } else {
        connectionStatusDiv.textContent = 'Disconnected from SpaceTimeDB';
        connectionStatusDiv.className = 'status-disconnected';
    }
}

// --- Main Connection Logic ---
async function initializeSpacetimeDB() {
    console.log(`Attempting to connect to SpaceTimeDB at ws://${SPACETIMEDB_HOS
T}:${SPACETIMEDB_PORT}`);

    // Step 1: Connect to the SpaceTimeDB server
    try {
        await SpacetimeDBClient.connect(
            SPACETIMEDB_HOST,
            SPACETIMEDB_PORT,
            SPACETIMEDB_DB_NAME,
            // You can provide an existing Identity or let the SDK create one
            Identity.fromHexString(localStorage.getItem('spacetime_identity') |
| undefined)
        );
        console.log('Connected to SpaceTimeDB!');
        updateConnectionStatus(true);

        // Store identity for persistence (optional, but good for user
recognition)
        localStorage.setItem('spacetime_identity', SpacetimeDBClient.identity.t
oHexString());
```

```
    } catch (error) {
        console.error('Failed to connect to SpaceTimeDB:', error);
        updateConnectionStatus(false);
        // Implement retry logic or alert user
        return;
    }

    // Step 2: Subscribe to the 'Message' table
    // The SDK will automatically handle initial sync and subsequent updates
    SpacetimeDBClient.subscribe([
        { tableName: 'Message' }
    ]);
    console.log('Subscribed to "Message" table.');

    // Step 3: Handle initial data and real-time updates
    // The `on` method allows us to listen to various events from the SDK.
    // `initial_sync` fires when the client first connects and receives all
subscribed data.
    SpacetimeDBClient.on("initial_sync", () => {
        console.log("Initial sync received!");
        // Get all messages after initial sync and render them
        const messages = SpacetimeDBClient.getEntities('Message');
        messages.sort((a, b) => Number(a.timestamp) - Number(b.timestamp)); //
Sort by timestamp
        messages.forEach(msg => appendMessage(msg.sender, msg.text, msg.timesta
mp));
    });

    // `update` fires for every change (insert, update, delete) to any
subscribed table
    SpacetimeDBClient.on("update", ({ events }) => {
        // We iterate through events to find changes to our 'Message' table
        events.forEach(event => {
            if (event.table_name === 'Message' && event.event_type ===
'insert') {
                const newMessage = event.row_new;
                // row_new contains the new state of the inserted row
                appendMessage(newMessage.sender, newMessage.text, newMessage.ti
mestamp);
            }
            // You could also handle 'update' or 'delete' events here if needed
        });
    });

    // Handle disconnection
    SpacetimeDBClient.on("disconnect", () => {
        console.warn("Disconnected from SpaceTimeDB. Attempting to
reconnect...");
        updateConnectionStatus(false);
        // You might want to implement a more robust reconnection strategy here
        setTimeout(initializeSpacetimeDB, 5000); // Try to reconnect after 5
seconds
    });

    // Step 4: Handle form submission to send messages (call reducer)
    messageForm.addEventListener('submit', async (event) => {
        event.preventDefault(); // Prevent default form submission

        const sender = usernameInput.value.trim();
        const text = messageInput.value.trim();

        if (sender && text) {
```

```
            console.log(`Calling reducer 'create_message' with sender: $
{sender}, text: ${text}`);
            try {
                // Call the server-side reducer
                await SpacetimeDBClient.call('create_message', sender, text);
                messageInput.value = ''; // Clear message input after sending
            } catch (error) {
                console.error('Error calling create_message reducer:', error);
                alert('Failed to send message. See console for details.');
            }
        } else {
            alert('Please enter your name and a message.');
        }
    });

    // Set a default username if not already set
    if (!usernameInput.value) {
        usernameInput.value = `Guest${Math.floor(Math.random() * 1000)}`;
    }
}

// Kick off the connection process when the script loads
initializeSpacetimeDB();
```

Explanation of `script.js`:

1. **`import { SpacetimeDBClient, Identity } from '@clockworklabs/spacetimedb-sdk';`**: We import the necessary classes from the SDK. `SpacetimeDBClient` is our main entry point, and `Identity` is used for client identification.
2. **Configuration**: `SPACETIMEDB_HOST`, `SPACETIMEDB_PORT`, and `SPACETIMEDB_DB_NAME` must match your running SpaceTimeDB instance.
3. **`SpacetimeDBClient.connect(...)`**: This is the first crucial step. It establishes the WebSocket connection.
    - `Identity.fromHexString(localStorage.getItem('spacetime_identity') || undefined)`: This line attempts to retrieve a previously saved client identity from `localStorage`. If found, the client will reconnect with the same identity, allowing SpaceTimeDB to recognize it. If not, a new identity is generated. This is great for persistent user sessions without full authentication (yet!).
4. **`SpacetimeDBClient.subscribe([{ tableName: 'Message' }])`**: Once connected, we tell SpaceTimeDB which tables we are interested in. Here, we subscribe to the `Message` table. This means the client will receive the initial data for this table and all subsequent changes.
5. **`SpacetimeDBClient.on("initial_sync", ...)`**: This event listener fires once when the client first receives the complete initial state of all subscribed

tables. We use `SpacetimeDBClient.getEntities('Message')` to fetch all current messages and render them.

6. **`SpacetimeDBClient.on("update", ...)`**: This is where the real-time magic happens! This event fires every time there's a change in any subscribed table. The `events` array contains detailed information about what changed (e.g., `event_type: 'insert'`, `table_name: 'Message'`, `row_new` with the inserted data). We filter for new `Message` inserts and append them to our chat display.
7. **`SpacetimeDBClient.on("disconnect", ...)`**: A simple handler for when the connection drops, attempting to reconnect after a delay.
8. **Form Submission (`messageForm.addEventListener`)**:
    - When the form is submitted, we prevent the default browser behavior.
    - `SpacetimeDBClient.call('create_message', sender, text)`: This is how we invoke our server-side reducer. The first argument is the reducer's name (as defined in `reducers.stdb`), and subsequent arguments are the parameters it expects, in order. The SDK handles sending this over the WebSocket and waiting for confirmation.
    - After a successful call, we clear the message input.

Step 4: Serve the Frontend

To run this, you need a simple web server. You can use Python's built-in server or Node.js `http-server`.

Using Node.js `http-server`:

1. **Install `http-server` (if you don't have it):**
   `bash npm install -g http-server`
2. **Run the server from your `chat-frontend` directory:** `bash http-server .` This will serve your `index.html` on `http://localhost:8080` (or another port).

Step 5: Test Your Real-time Chat

1. Ensure your SpaceTimeDB instance is running (e.g., `spacetime db dev`).
2. Open your browser to `http://localhost:8080`.
3. Open a second browser tab or window to the same address.
4. Type messages in either window. You should see them appear in real-time in both windows!

Congratulations! You've successfully built your first real-time web application integrated with SpaceTimeDB.

Integration with Game Engines (Conceptual)

While a full Unity example is beyond the scope of a single chapter, the principles are identical for game engines using the C# SDK.

1. **Install C# SDK:** Add the SpaceTimeDB C# SDK package to your Unity project. This is typically done via a `.unitypackage` or NuGet.
2. **Connect:** In a MonoBehaviour script (e.g., `GameManager.cs`), use `SpacetimeDBClient.ConnectAsync()` to establish the connection to your SpaceTimeDB server.
3. **Subscribe:** Call `SpacetimeDBClient.Subscribe()` to specify which tables your game needs to observe (e.g., `Player`, `Enemy`, `Projectile`).
4. **Handle Updates:**
    - Register callbacks for `SpacetimeDBClient.onUpdate` or specific table change events.
    - When an update arrives, use the data to update your game's entities: move player characters, spawn new objects, update health bars, etc.
    - Remember to handle updates on the main Unity thread if you're modifying UI or game objects, as SpaceTimeDB events might arrive on a background thread.
5. **Call Reducers:** When a player performs an action (e.g., "move character," "shoot weapon"), call the corresponding server-side reducer using `SpacetimeDBClient.CallReducerAsync("move_player", x, y)`. The game logic within the reducer will validate the action and update the database, which then propagates back to all clients.

The key is mapping the real-time data from SpaceTimeDB to your game's visual and logical components, and mapping player input to reducer calls.

Mini-Challenge: Enhance the Chat with User Colors

Let's make our chat a bit more colorful!

Challenge: Modify the chat application so that each user (identified by their `sender` name) has a consistent, randomly assigned color for their messages. This color should be determined when they first send a message and remain the same for that user across all their messages and for all clients.

Hint: * You'll need a new table in your SpaceTimeDB schema (e.g., `UserColor`) to store which `sender` has which `color`. * Your `create_message` reducer will need to check if a color already exists for the `sender`. If not, it should generate a random color (e.g., a hex string like "#RRGGBB") and store it in the `UserColor` table before inserting the message. * Your client-side code will need to subscribe to this new `UserColor` table and use that information when rendering messages. You'll need to fetch the color for each message's sender.

What to observe/learn: * How to extend your SpaceTimeDB schema and reducer logic to manage additional shared state. * How to subscribe to multiple tables from the client. * How to combine data from different subscribed tables on the client to enrich the UI. * The deterministic nature of reducers: a random color generated within a reducer will be the same for everyone.

Common Pitfalls & Troubleshooting

1. **Connection Issues (`Failed to connect to SpaceTimeDB`):**

- **Check SpaceTimeDB Server:** Is your `spacetime db dev` instance actually running?
- **Host/Port Mismatch:** Ensure `SPACETIMEDB_HOST` and `SPACETIMEDB_PORT` in your `script.js` exactly match where your SpaceTimeDB server is listening.
- **Firewall:** Local firewalls can sometimes block WebSocket connections.
- **Module Name:** Double-check `SPACETIMEDB_DB_NAME` matches the name of your SpaceTimeDB module.

1. **Reducer Call Errors (`Error calling create_message reducer`):**

- **Reducer Name Mismatch:** Is the string passed to `SpacetimeDBClient.call()` an exact match for your reducer function's name in `reducers.stdb`? (e.g., `create_message` vs `CreateMessage`). Reducer names are case-sensitive.
- **Argument Mismatch:** Do the number and types of arguments passed from the client match the reducer's signature in Rust? (e.g., `create_message(sender: String, text: String)` expects two strings).

- **Reducer Logic Errors:** Check your SpaceTimeDB server logs. If the reducer panics or returns an error, it will be reported there, and the client will receive an error.

1. **No Real-time Updates:**

- **Subscription Missing/Incorrect:** Did you call `SpacetimeDBClient.subscribe()` for the correct table names?
- **Event Listener Order:** Ensure your `SpacetimeDBClient.on("initial_sync", ...)` and `SpacetimeDBClient.on("update", ...)` listeners are registered before the `initial_sync` or `update` events might occur (i.e., immediately after `connect`).
- **Table Changes Not Occurring:** Are your reducers actually modifying the tables you expect them to? Check the SpaceTimeDB CLI `db dump` or `db watch` commands.

1. **Identity Persistence Issues:**
    - If `localStorage` isn't working or is cleared, the client will get a new `Identity` on each connection. This might be desired, but if you expect persistent recognition without full authentication, ensure `localStorage.setItem` and `localStorage.getItem` are working correctly.

Remember, the SpaceTimeDB server logs are your best friend for debugging server-side reducer issues, and your browser's developer console is essential for client-side debugging.

Summary

Phew! You've come a long way. In this chapter, we explored the crucial topic of integrating your SpaceTimeDB backend with frontend applications. Here's a quick recap of what we covered:

- **Client SDKs:** These libraries (like `@clockworklabs/spacetimedb-sdk` for JavaScript/TypeScript) simplify connecting, subscribing, and calling reducers.
- **Real-time Synchronization:** We learned how SpaceTimeDB pushes initial data and subsequent updates over WebSockets, enabling instant UI reactions.
- **Reducer Invocation:** Clients interact with the database by calling server-side reducers, ensuring centralized logic, determinism, and security.

- **Practical Web Integration:** We built a hands-on real-time chat application, demonstrating how to connect, subscribe to table changes, process updates, and invoke reducers from a vanilla JavaScript frontend.
- **Game Engine Concepts:** We conceptually discussed how similar principles apply to integrating with game engines like Unity using their respective SDKs.
- **Troubleshooting:** We looked at common issues like connection failures, reducer call errors, and missing real-time updates.

You now have the foundational knowledge to build truly interactive, real-time user experiences powered by SpaceTimeDB. The ability to seamlessly connect your frontend to a globally consistent, event-driven backend opens up a world of possibilities for multiplayer games, collaborative tools, and dynamic dashboards.

What's Next?

In the next chapter, we'll delve into more advanced aspects of SpaceTimeDB, focusing on topics like authentication, security models, and how to manage user access in your real-time applications.

References

- SpaceTimeDB Official Website
- SpaceTimeDB Documentation
- SpaceTimeDB GitHub Repository
- SpaceTimeDB JavaScript/TypeScript SDK (Refer to the `sdk` crate within the main repo for the JS/TS SDK's source and usage)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 17

Chapter 9: Ensuring Consistency: Concurrency, Transactions, and Determinism

Chapter 9: Ensuring Consistency: Concurrency, Transactions, and Determinism

Welcome to Chapter 9! So far, we've explored how SpaceTimeDB combines database, backend logic, and real-time synchronization. We've built schemas, written reducers, and seen how clients react to state changes. But as applications grow and multiple users interact simultaneously, a critical question arises: How does SpaceTimeDB keep everything consistent and reliable?

In this chapter, we're going to pull back the curtain on some of SpaceTimeDB's most powerful, yet often invisible, features: **concurrency control**, **transactional integrity**, and **deterministic execution**. These are the bedrock upon which SpaceTimeDB builds its promise of "multiplayer at the speed of light." Understanding these concepts is vital for designing robust, bug-free real-time systems that behave predictably, no matter how many users are interacting at once. Get ready to explore the "why" and "how" behind SpaceTimeDB's impressive consistency guarantees!

The Challenge of Concurrency in Real-time Systems

Imagine two players in a game trying to pick up the same rare item at precisely the same moment. Or two collaborators in a document editor attempting to modify the same paragraph. In traditional distributed systems, handling these "concurrent" operations without introducing data corruption or inconsistent states is incredibly difficult. You often face race conditions, deadlocks, and complex locking mechanisms.

SpaceTimeDB is designed from the ground up to solve this. It provides strong consistency guarantees, meaning that all connected clients eventually see the same, correct state, and all operations are processed reliably. But how does it achieve this? The answer lies in its unique approach to transactions and deterministic execution.

Transactions: The Foundation of Reliability

In database systems, a **transaction** is a sequence of operations performed as a single logical unit of work. The classic way to describe reliable transactions is through the ACID properties:

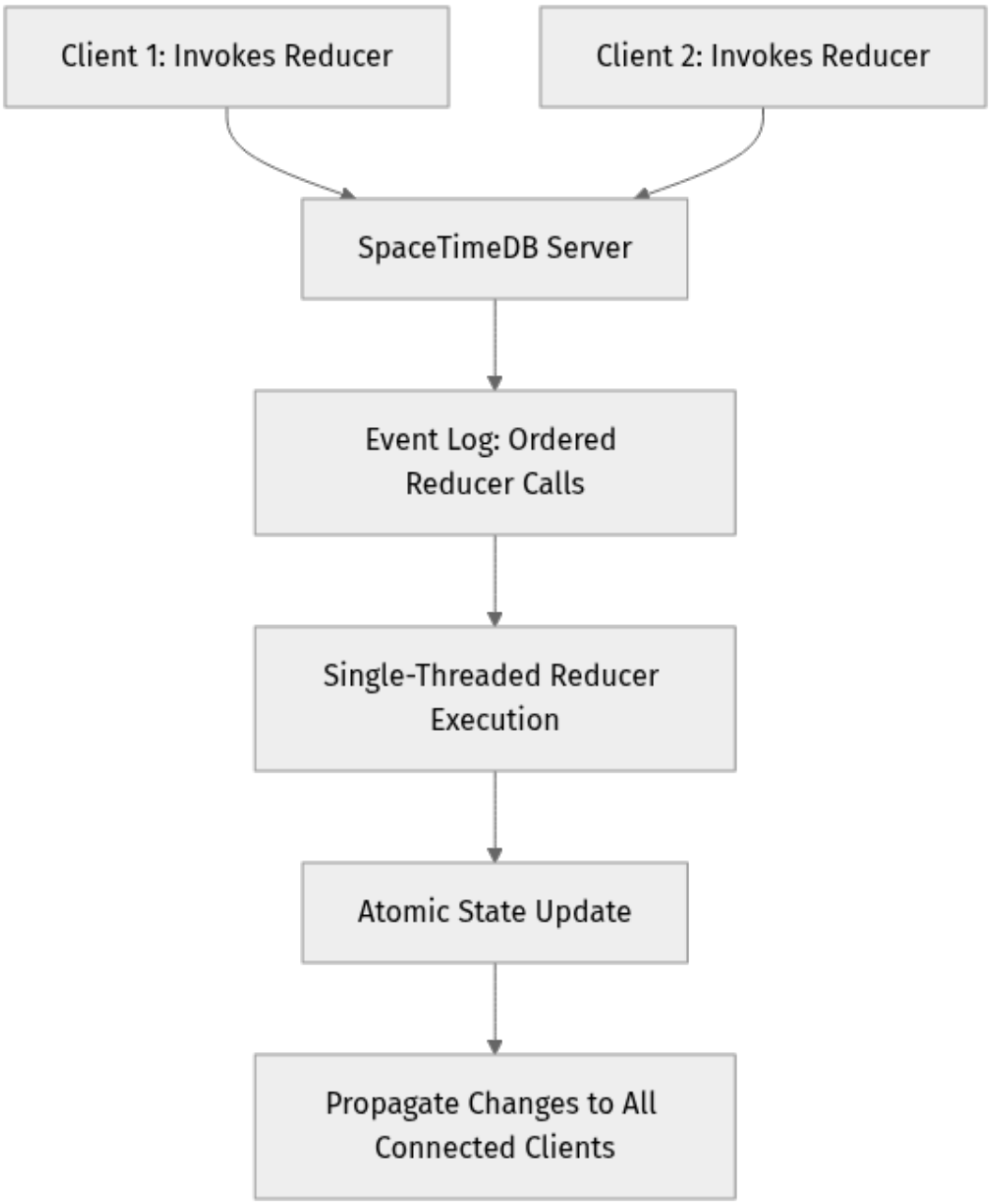
- **Atomicity**: A transaction is an indivisible unit of work. Either all of its operations are completed successfully, or none of them are. There's no "half-finished" state.
- **Consistency**: A transaction brings the database from one valid state to another. It ensures that any data written to the database must be valid according to all defined rules (like schema constraints).
- **Isolation**: Concurrent transactions execute in such a way that they appear to run sequentially. The intermediate state of one transaction is not visible to other transactions.
- **Durability**: Once a transaction is committed, its changes are permanent and survive any subsequent system failures.

SpaceTimeDB's **reducers** are inherently transactional. When you call a reducer, SpaceTimeDB treats its entire execution as an atomic operation. Let's see how this works.

SpaceTimeDB's Reducers as Atomic Operations

Every time a client calls a reducer, that call is added to an ordered **event log** on the SpaceTimeDB server. The server then processes these reducer calls one by one, in a strictly serial fashion. This single-threaded execution of reducers is key to its consistency model.

Here's a simplified view of this internal flow:



1. **Client Invocation**: Multiple clients can call reducers concurrently.
2. **Server Receives**: The SpaceTimeDB server receives these calls.
3. **Event Log**: Each reducer call is appended to an ordered, immutable event log. This log is the source of truth for all state changes.
4. **Single-Threaded Reducer Execution**: SpaceTimeDB's core logic processes entries from the event log one at a time. This means that even if 100 clients try to update something simultaneously, their reducer calls will be executed sequentially by the server.
5. **Atomic State Update**: The execution of a reducer is an atomic operation. If the reducer successfully completes, all its changes are applied to the

database state. If it panics or returns an error, none of its changes are applied, and the state remains as it was before the reducer started. This guarantees atomicity.

6. **Propagate Changes**: Once the state is updated, SpaceTimeDB deterministically calculates the new diffs and propagates them to all subscribed clients in real-time.

This serial execution of reducers, combined with the event log, is what gives SpaceTimeDB its strong isolation and consistency guarantees. It removes the need for you, the developer, to worry about complex locks or race conditions within your reducer logic.

Determinism: Predictable Outcomes, Everywhere

Determinism in this context means that given the same starting state and the same sequence of reducer calls, SpaceTimeDB will always produce the exact same final state. No matter when or where those reducer calls are executed, the outcome is predictable.

Why is this important?

- **Consistency**: It ensures that every SpaceTimeDB replica (if in a distributed setup) and every client can derive the same "truth" from the event log.
- **Debugging**: Makes it easier to reproduce bugs, as the sequence of events leading to an issue is recorded and can be replayed.
- **Trust**: You can trust that the logic you write in your reducers will always yield the expected results, regardless of network latency or concurrent client actions.

How Reducers Ensure Determinism

For SpaceTimeDB to be deterministic, your reducers must also be deterministic. This means:

- **No Randomness**: Reducers should not use random number generators. If randomness is needed (e.g., for rolling dice in a game), the random seed or the random value itself should be passed into the reducer as an argument from the client, or generated deterministically based on input.
- **No External Side Effects**: Reducers should not make external API calls, interact with the file system, or query external databases. Their only input should be their arguments and the current SpaceTimeDB state, and their only output should be modifications to the SpaceTimeDB state.

- **Pure Functions (mostly)**: Think of reducers as pure functions: given the same inputs (current state + reducer arguments), they always produce the same output (new state).

SpaceTimeDB's Rust environment for reducers helps enforce this. While Rust is a powerful language, the SpaceTimeDB runtime restricts certain operations within reducers to maintain determinism.

Step-by-Step Implementation: Building a Transactional Inventory System

Let's build a simple inventory system for a game where players can "pick up" items. We'll ensure that an item can only be picked up if it still exists and that the player's inventory is updated atomically.

First, let's define our schema in `schema.stdb`:

```
// schema.stdb
table items {
    item_id: u32,
    name: String,
    owner_id: Option<u32>, // None if on ground, Some(player_id) if picked up
    location_x: f32,
    location_y: f32,
}

table players {
    player_id: u32,
    name: String,
    // Other player properties
}
```

Now, let's write a reducer in `lib.rs` that handles picking up an item. This reducer will demonstrate SpaceTimeDB's transactional guarantees.

Assume you have a `Player` and `Item` `Row` structs generated by `stdb generate`.

```
// src/lib.rs

// Import necessary SpaceTimeDB types
use spacetimedb::{
    spacetimedb,
    Identity,
    ReducerContext,
    table::{Table, TableWith
    }
};

// Import our generated schema types
use crate::items::Item;
use crate::players::Player;

// --- Reducers ---

#[spacetimedb(reducer)]
pub fn create_player(ctx: ReducerContext, player_id: u32, name: String) -> Resu
lt<(), String> {
    if Player::filter_by_player_id(player_id).count() > 0 {
        return Err("Player with this ID already exists".to_string());
    }
    Player::insert(Player {
        player_id,
        name,
        // ... other default player fields
    });
    Ok(())
}

#[spacetimedb(reducer)]
pub fn create_item(ctx: ReducerContext, item_id: u32, name: String,
location_x: f32, location_y: f32) -> Result<(), String> {
    if Item::filter_by_item_id(item_id).count() > 0 {
        return Err("Item with this ID already exists".to_string());
    }
    Item::insert(Item {
        item_id,
        name,
        owner_id: None, // Initially on the ground
        location_x,
        location_y,
    });
    Ok(())
}

#[spacetimedb(reducer)]
pub fn pick_up_item(ctx: ReducerContext, player_id: u32, item_id: u32) -> Resul
t<(), String> {
    // 1. Check if the player exists
    let player_exists = Player::filter_by_player_id(player_id).count() > 0;
    if !player_exists {
        return Err(format!("Player with ID {} not found.", player_id));
    }

    // 2. Try to find the item
    // We use `Table::find` which returns an `Option<Item>`
    let mut item_opt = Item::filter_by_item_id(item_id).into_iter().next();

    // 3. Check if item exists and is not already owned
```

```
    match item_opt {
        Some(mut item) => {
            if item.owner_id.is_some() {
                // This item is already owned!
                return Err(format!
("Item with ID {} is already owned by another player.", item_id));
            }

            // 4. Update the item's owner
            item.owner_id = Some(player_id);
            item.update(); // Update the item in the database

            // If we wanted to add it to a player's inventory directly, we'd
update `Player` table here
            // For now, `owner_id` on the item suffices.

            Ok(())
        }
        None => {
            // Item not found
            Err(format!("Item with ID {} not found.", item_id))
        }
    }
}
```

Explanation of the `pick_up_item` Reducer:

1. **Input**: The reducer takes `player_id` and `item_id` as arguments.
2. **Player Existence Check**: It first verifies that the `player_id` corresponds to an existing player. If not, it returns an error, and the transaction is aborted.
3. **Item Retrieval**: It attempts to find the `Item` by its `item_id`.
4. **Ownership Check**: Crucially, it checks if the item already has an `owner_id`. If `item.owner_id.is_some()` is true, it means another player has already picked it up (or is in the process of doing so). In this case, it returns an error.
5. **Atomic Update**: If the item exists and is not owned, its `owner_id` is updated to the `player_id`, and `item.update()` is called.

How Concurrency is Handled Here:

Imagine two clients, Player A and Player B, both call `pick_up_item` for `item_id = 123` at almost the exact same time.

- Both calls hit the SpaceTimeDB server.
- They are added to the event log: `[pick_up_item(PlayerA, 123), pick_up_item(PlayerB, 123)]` (or vice-versa).

- **First Reducer Execution (e.g., PlayerA)**:
  - `pick_up_item(PlayerA, 123)` runs.
  - It finds `item_id = 123`.
  - `item.owner_id` is `None`.
  - `item.owner_id` is set to `Some(PlayerA)`.
  - `item.update()` commits this change.
  - The reducer returns `Ok(())`.
- **Second Reducer Execution (PlayerB)**:
  - `pick_up_item(PlayerB, 123)` runs.
  - It finds `item_id = 123`.
  - **Crucially, at this point, *`item.owner_id`* is already *`Some(PlayerA)`* because the previous reducer call completed and updated the state.**
  - The `if item.owner_id.is_some()` condition evaluates to true.
  - The reducer returns `Err("Item with ID 123 is already owned...")`.
  - The transaction for Player B's attempt is aborted, and no state changes are applied.

The outcome is perfectly consistent: only one player successfully picks up the item, and the other receives an appropriate error. This happens without you writing any explicit locking code, thanks to SpaceTimeDB's single-threaded, transactional reducer execution.

Mini-Challenge: Transferring Gold

Let's expand on our example. Imagine players have a `gold` balance. Create a reducer that allows a player to transfer gold to another player. This operation must be atomic: either both the sender's balance is debited and the receiver's credited, or neither happens.

Challenge:

1. Add a `gold: u32` field to your `players` table in `schema.stdb`.
2. Write a new reducer `transfer_gold(ctx: ReducerContext, sender_id: u32, receiver_id: u32, amount: u32)` that:
   - Verifies both sender and receiver exist.
   - Checks if the sender has enough gold.

    - If all checks pass, it subtracts `amount` from the sender's gold and adds `amount` to the receiver's gold.
    - If any check fails, the entire operation should fail (return an `Err`), and no gold should be transferred.

Hint: Remember that `Table::filter_by_...` returns an iterator. Use `.into_iter().next()` to get an `Option<Row>`, and then `unwrap_or_else` or `match` to handle the `None` case. You'll need to make the retrieved player structs mutable (`mut player`) to update their fields and then call `player.update()`.

What to Observe/Learn:

- How multiple database operations (reading, updating two different rows) are treated as a single atomic unit within a reducer.
- The importance of early `return Err` statements to abort a transaction if conditions aren't met.
- How SpaceTimeDB's internal mechanisms ensure that even if two players try to transfer gold simultaneously, the final balances will always be correct, without double-spending or creating gold out of thin air.

Common Pitfalls & Troubleshooting

1. **Non-Deterministic Reducers**:

- **Pitfall**: Introducing randomness (e.g., `rand::thread_rng().gen_range(0..10)`) or external API calls within a reducer.
- **Troubleshooting**: SpaceTimeDB's reducer runtime is designed to prevent many non-deterministic operations. If you attempt one, you'll likely get a compilation error or a runtime panic indicating an unsupported operation. If you need randomness, generate it on the client and pass it as a reducer argument, or implement a deterministic pseudo-random number generator based on a known seed within your reducer. For external data, fetch it on the client and pass it in, or use SpaceTimeDB's upcoming external module features (check official docs for latest on this). 2. **Over-reliance on Client-Side State for Critical Logic**:
- **Pitfall**: Performing complex validation or game logic solely on the client and only sending a simple update command to the reducer. This makes your application vulnerable to cheating or inconsistent states if the client is compromised or has stale data.

- **Troubleshooting**: Always perform critical validation and state-changing logic inside reducers. SpaceTimeDB's server-side execution is the single source of truth and is protected from client-side manipulation. 3. **Complex Reducers Leading to Long Execution Times**:
- **Pitfall**: While reducers are atomic, they are also single-threaded. A very long-running or computationally intensive reducer can block other incoming reducer calls, leading to perceived latency for users.
- **Troubleshooting**: Keep reducers lean and focused on state updates. Offload heavy computations to client-side logic or separate serverless functions that then call a SpaceTimeDB reducer with the final, validated outcome. Monitor reducer execution times in production.

Summary

In this chapter, we've explored the fundamental principles of consistency in SpaceTimeDB:

- **Concurrency**: SpaceTimeDB addresses concurrency challenges by processing reducer calls in a strictly serial, ordered fashion, avoiding race conditions.
- **Transactions**: Reducers execute atomically, ensuring that all operations within them either succeed entirely or fail entirely, upholding the ACID properties (especially Atomicity, Consistency, and Isolation).
- **Determinism**: The SpaceTimeDB engine and well-written reducers guarantee that given the same starting state and event sequence, the final state will always be identical, which is crucial for reliability and debugging.
- We saw how to leverage these guarantees by writing a `pick_up_item` reducer that correctly handles concurrent attempts without explicit locking.

By understanding these core mechanics, you can confidently build complex, real-time applications knowing that SpaceTimeDB provides a robust and consistent foundation.

Next, we'll delve into performance optimization and scaling strategies, learning how to make your SpaceTimeDB applications not just consistent, but also lightning-fast and capable of handling massive loads!

References

1. **SpaceTimeDB Official Documentation**: The primary source for all SpaceTimeDB concepts and API details.
    - https://spacetimedb.com/docs
2. **SpaceTimeDB GitHub Repository**: Explore the source code and latest releases (v2.x as of March 2026).
    - https://github.com/clockworklabs/SpacetimeDB
3. **ACID Properties Explained**: A good general overview of database transaction properties.
    - https://en.wikipedia.org/wiki/ACID

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.