Project Summary: Rust egui Script Manager

Overview

This project is a desktop application built with Rust and egui, using Prisma for SQLite database management. It allows users to manage scripts organized in folders, with features for creating, editing, renaming, copying, and executing scripts.

What We Accomplished

1. Fixed Initial Compilation Error

- **Issue**: `no field 'scripts_folder_id' on type &std::vec::Vec<rel_scriptsfolder_shellscript::Data>`
- **Root Cause**: Incorrect handling of Prisma's `Option<Vec<T>>` return types from `.with()` relations.
- **Solution**: Updated filter logic in `folder_event_handler.rs` to properly handle optional collections.
- **Lesson**: Prisma relations return `Option<Vec<T>>`, requiring careful unwrapping and iteration.

2. Implemented Script Loading for Folders

- **Feature**: When a folder is selected, load and display associated scripts.
- **Implementation**: Used Prisma's association tables (`rel_scriptsfolder_shellscript`) for many-to-many relationships.
- **Code Changes**: Modified event handlers to query and filter scripts by folder ID.

3. Added Script Management Features

- **Edit Command**: Modal window to edit script commands with syntax highlighting.
- **Copy Command**: One-click copying of script text to clipboard.
- **Execute Command**: Run scripts via terminal with async output logging.
- **Rename Script**: Inline renaming with database updates.
- **Add Script**: Create new scripts in selected folders.

4. Enhanced UI/UX

- **Multiline Display**: Scripts shown in read-only TextEdit with proper line breaks.
- **Theme Integration**: Used `ui.visuals().faint_bg_color` for consistent, adaptive backgrounds.
- **Button Layout**: Organized buttons (Rename, Execute, Edit, Copy) with logical positioning.
- **Visual Improvements**: Lighter backgrounds for script groups and command areas.

5. Database Operations

- **Commands Added**:
  - `UpdateScriptToFolder`: Update script command text.
  - `UpdateScriptNameToFolder`: Rename scripts.
- **Events Added**: `ScriptUpdated` for UI refresh after changes.
- **Async Handling**: All DB operations use Tokio tasks for non-blocking UI.

6. Debugging and Logging

- **Dispatch Logging**: Added `println!` in `dispatch_folder_command` and `dispatch_folder_event` to log all enum dispatches.
- **Execution Logging**: Terminal command execution outputs logged to console.

Key Technologies and Concepts Learned

Rust and Async Programming

- **Tokio Runtime**: Used for async database operations and command execution.
- **Spawn Tasks**: `spawn_task()` for background operations without blocking UI.
- **Error Handling**: Proper `Result` handling in async contexts.

egui UI Framework

- **Widgets**: TextEdit, Button, Label, Frame, ScrollArea.
- **Layout**: Horizontal/vertical layouts, right-to-left button positioning.
- **Styling**: Custom frames, theme colors (`faint_bg_color`), background fills.
- **Event Handling**: Click detection and state management.

Prisma ORM

- **Relations**: Handling many-to-many relationships with association tables.
- **Queries**: Using `.with()` for eager loading, filtering with `.find_many()`.
- **Updates**: Atomic updates with `.update_many()` and field setters.
- **Type Safety**: Strong typing with generated structs.

Event-Driven Architecture

- **Commands**: User actions trigger commands (e.g., `FolderCommand::UpdateScriptToFolder`).
- **Events**: Commands produce events (e.g., `FolderEvent::ScriptUpdated`) for UI updates.
- **Dispatch System**: Centralized event/command bus with crossbeam channels.

Database Design

- **Schema**: Scripts, folders, and association tables for relationships.
- **Migrations**: Embedded Prisma migrations for schema evolution.
- **SQLite**: Embedded database for portability.

Best Practices

- **Separation of Concerns**: Domain logic (commands/events) separate from UI.
- **Immutability**: Using `Arc` for shared state, careful mutation.
- **Debugging**: Strategic logging for event flow and async operations.
- **Theme Awareness**: Using egui's visual system for consistent styling.

File Changes Summary

- `src/domain/folder/folder_command_handler.rs`: Added new commands and handlers.

- `src/domain/folder/folder_event_handler.rs`: Added event handling for script updates.
- `src/component/scripts_col.rs`: Major UI updates for script display and interaction.
- `src/lib.rs`: Added terminal execution, dispatch logging, and helper functions.
- `src/prisma.rs`: Generated types (no manual changes).

Lessons Learned

1. **Prisma Relations**: Always handle `Option<Vec<T>>` properly when using `.with()`.
2. **egui Styling**: Use `Frame` with custom fills for backgrounds, leverage theme colors.
3. **Async UI**: Keep UI responsive by offloading work to tasks.
4. **Event System**: Commands → Events → UI updates provide clean architecture.
5. **Debugging**: Logging dispatches helps trace application flow.

This project demonstrates building a full-featured desktop app with modern Rust patterns, from database integration to polished UI.

Code Examples

Here are practical code examples demonstrating the key patterns and concepts used in this project:

1. Shared State with Arc<RwLock>

```
use std::sync::{Arc, RwLock};

#[derive(Debug, Clone)]
struct Folder {
    id: i32,
    name: String,
}

struct AppState {
    folders: Arc<RwLock<Vec<Folder>>>,
    selected_folder: Arc<RwLock<Option<i32>>>,
}

impl AppState {
    fn new() -> Self {
        Self {
            folders: Arc::new(RwLock::new(Vec::new())),
            selected_folder: Arc::new(RwLock::new(None)),
        }
    }

    // Reading shared state
    fn get_folders(&self) -> Vec<Folder> {
        self.folders.read().unwrap().clone()
    }

    // Writing to shared state
    fn add_folder(&self, folder: Folder) {
        self.folders.write().unwrap().push(folder);
```

```
    }
}
```

2. Async Database Operations with Tokio

```
async fn load_folders_from_db() -> Result<Vec<Folder>, Box<dyn
std::error::Error>> {
    // Simulate async DB call (in real code, this would be Prisma)
    tokio::time::sleep(tokio::time::Duration::from_millis(100)).await;

    Ok(vec![
        Folder { id: 1, name: "Scripts".to_string() },
        Folder { id: 2, name: "Tools".to_string() },
    ])
}

async fn save_folder_to_db(folder: &Folder) -> Result<(), Box<dyn
std::error::Error>> {
    println!("Saving folder: {:?}", folder);
    tokio::time::sleep(tokio::time::Duration::from_millis(50)).await;
    Ok(())
}
```

3. Event/Command System with Crossbeam Channels

```
use crossbeam::channel::{self, Receiver, Sender};

#[derive(Debug)]
enum AppCommand {
    AddFolder(Folder),
    SelectFolder(i32),
}

#[derive(Debug)]
enum AppEvent {
    FolderAdded(Folder),
    FolderSelected(i32),
}

struct EventSystem {
    command_sender: Sender<AppCommand>,
    command_receiver: Receiver<AppCommand>,
    event_sender: Sender<AppEvent>,
    event_receiver: Receiver<AppEvent>,
}

impl EventSystem {
    fn new() -> Self {
        let (cmd_tx, cmd_rx) = channel::unbounded();
```

```
        let (evt_tx, evt_rx) = channel::unbounded();

        Self {
            command_sender: cmd_tx,
            command_receiver: cmd_rx,
            event_sender: evt_tx,
            event_receiver: evt_rx,
        }
    }

    fn dispatch_command(&self, command: AppCommand) {
        println!("Dispatching command: {:?}", command);
        let _ = self.command_sender.send(command);
    }
}
```

4. Background Tasks with Tokio Spawn

```
async fn command_handler(
    mut command_rx: Receiver<AppCommand>,
    event_tx: Sender<AppEvent>,
    state: Arc<AppState>,
) {
    while let Ok(command) = command_rx.recv() {
        match command {
            AppCommand::AddFolder(folder) => {
                // Async DB operation
                if let Err(e) = save_folder_to_db(&folder).await {
                    eprintln!("Failed to save folder: {:?}", e);
                    continue;
                }

                // Update shared state
                state.add_folder(folder.clone());

                // Dispatch event
                let _ = event_tx.send(AppEvent::FolderAdded(folder));
            }
            AppCommand::SelectFolder(id) => {
                state.select_folder(id);
                let _ = event_tx.send(AppEvent::FolderSelected(id));
            }
        }
    }
}
```

5. Terminal Command Execution

```
async fn run_terminal_command(command: String) -> Result<String, Box<dyn
std::error::Error>> {
    let output = tokio::process::Command::new("sh")
        .arg("-c")
        .arg(&command)
        .output()
        .await?;

    let stdout = String::from_utf8_lossy(&output.stdout).to_string();
    let stderr = String::from_utf8_lossy(&output.stderr).to_string();

    println!("Executed: {}", command);
    if !stdout.is_empty() {
        println!("Output: {}", stdout);
    }
    if !stderr.is_empty() {
        eprintln!("Error: {}", stderr);
    }

    Ok(stdout)
}
```

6. Main Application Loop with Event Processing

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // Initialize shared state
    let app_state = Arc::new(AppState::new());

    // Initialize event system
    let event_system = Arc::new(EventSystem::new());

    // Start command handler in background
    let command_handler_state = Arc::clone(&app_state);
    let event_tx = event_system.event_sender.clone();
    let command_rx = event_system.command_receiver.clone();

    tokio::spawn(async move {
        command_handler(command_rx, event_tx,
command_handler_state).await;
    });

    // Initialize UI state
    let ui_state = UiState::new(Arc::clone(&app_state),
Arc::clone(&event_system));

    // Simulate application loop
    for _ in 0..5 {
        // Process events
        while let Ok(event) = event_system.event_receiver.try_recv() {
```

```
            println!("Received event: {:?}", event);
        }

        // Render UI
        ui_state.render();

        // Simulate frame delay
        tokio::time::sleep(tokio::time::Duration::from_millis(200)).await;
    }

    Ok(())
}
```

7. Unit Tests for Shared State and Events

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_shared_state() {
        let state = AppState::new();

        // Test writing
        let folder = Folder { id: 1, name: "Test".to_string() };
        state.add_folder(folder.clone());

        // Test reading
        let folders = state.get_folders();
        assert_eq!(folders.len(), 1);
        assert_eq!(folders[0].name, "Test");
    }

    #[test]
    fn test_event_system() {
        let event_system = EventSystem::new();

        // Test command dispatch
        let command = AppCommand::SelectFolder(42);
        event_system.dispatch_command(command);

        // Test event dispatch
        let event = AppEvent::FolderSelected(42);
        event_system.dispatch_event(event);

        // Verify messages were sent
        assert!(event_system.command_receiver.try_recv().is_ok());
        assert!(event_system.event_receiver.try_recv().is_ok());
    }
}
```

These examples show how to:

- Use `Arc<RwLock<T>>` for thread-safe shared state
- Handle async operations with Tokio
- Implement event-driven architecture with channels
- Run background tasks without blocking the UI
- Execute terminal commands asynchronously
- Write unit tests for concurrent code

The complete example code is also available in `code_examples.rs` in the project root.