Rust Integration Example

This document shows how to integrate the Spring Boot backend with your Tauri Rust application.

Overview

Instead of direct database access via Prisma, the Rust backend will:

1. Launch the Spring Boot backend on startup
2. Make HTTP requests to Spring Boot APIs
3. Return results to the Tauri frontend

Step 1: Add Dependencies to Cargo.toml

Add to `src-tauri/Cargo.toml`:

```
[dependencies]
# Existing dependencies...
tokio = { version = "1", features = ["full"] }
reqwest = { version = "0.11", features = ["json", "blocking"] }
serde_json = "1.0"
```

Step 2: Create HTTP Client Module

Create `src-tauri/src/http_client/mod.rs`:

```
use reqwest::blocking::Client;
use serde::{Deserialize, Serialize};

const BASE_URL: &str = "http://localhost:8080/api";

pub struct BackendClient {
    client: Client,
}

impl BackendClient {
    pub fn new() -> Self {
        Self {
            client: Client::new(),
        }
    }

    pub fn get_all_folders(&self) -> Result<Vec<Folder>, String> {
        let url = format!("{}/folders", BASE_URL);
        self.client
            .get(&url)
            .send()
```

```
            .map_err(|e| format!("Request failed: {}", e))?
            .json::<Vec<Folder>>()
            .map_err(|e| format!("Failed to parse response: {}", e))
    }

    pub fn create_folder(&self, name: &str) -> Result<Folder, String> {
        let url = format!("{}/folders", BASE_URL);
        let body = CreateFolderRequest {
            name: name.to_string(),
            ordering: 0,
            created_at: current_timestamp(),
            created_at_hk: current_timestamp_hk(),
        };

        self.client
            .post(&url)
            .json(&body)
            .send()
            .map_err(|e| format!("Request failed: {}", e))?
            .json::<Folder>()
            .map_err(|e| format!("Failed to parse response: {}", e))
    }

    pub fn update_folder(&self, id: i32, name: &str, ordering: i32) ->
Result<Folder, String> {
        let url = format!("{}/folders/{}", BASE_URL, id);
        let body = UpdateFolderRequest {
            name: name.to_string(),
            ordering,
            created_at: current_timestamp(),
            created_at_hk: current_timestamp_hk(),
        };

        self.client
            .put(&url)
            .json(&body)
            .send()
            .map_err(|e| format!("Request failed: {}", e))?
            .json::<Folder>()
            .map_err(|e| format!("Failed to parse response: {}", e))
    }

    pub fn delete_folder(&self, id: i32) -> Result<(), String> {
        let url = format!("{}/folders/{}", BASE_URL, id);
        self.client
            .delete(&url)
            .send()
            .map_err(|e| format!("Request failed: {}", e))?;
        Ok(())
    }

    // Similar methods for scripts and app state...
}
```

```rust
#[derive(Debug, Serialize, Deserialize)]
pub struct Folder {
    pub id: i32,
    pub name: String,
    pub ordering: i32,
    pub created_at: f64,
    pub created_at_hk: String,
}

#[derive(Debug, Serialize)]
struct CreateFolderRequest {
    name: String,
    ordering: i32,
    created_at: f64,
    created_at_hk: String,
}

#[derive(Debug, Serialize)]
struct UpdateFolderRequest {
    name: String,
    ordering: i32,
    created_at: f64,
    created_at_hk: String,
}

// Helper functions for timestamps
fn current_timestamp() -> f64 {
    use std::time::{SystemTime, UNIX_EPOCH};
    SystemTime::now()
        .duration_since(UNIX_EPOCH)
        .unwrap()
        .as_millis() as f64
}

fn current_timestamp_hk() -> String {
    use chrono::{DateTime, FixedOffset, Utc};
    let hk_offset = FixedOffset::east_opt(8 * 3600).unwrap();
    let now: DateTime<Utc> = Utc::now();
    now.with_timezone(&hk_offset)
        .format("%Y-%m-%d %H:%M:%S")
        .to_string()
}
```

Step 3: Launch Spring Boot on Startup

Add to `src-tauri/src/lib.rs`:

```rust
mod http_client;

use std::process::{Command, Child};
use std::sync::Mutex;
```

```rust
use std::thread;
use std::time::Duration;

// Global to hold the Spring Boot process
static SPRING_BOOT_PROCESS: Mutex<Option<Child>> = Mutex::new(None);

/// Start the Spring Boot backend
pub fn start_spring_boot_backend() -> Result<(), String> {
    println!("Starting Spring Boot backend...");

    #[cfg(debug_assertions)]
    {
        // Development mode: Launch via gradlew
        let child = Command::new("./gradlew")
            .arg("bootRun")
            .current_dir(../backend-spring")
            .stdout(std::process::Stdio::piped())
            .stderr(std::process::Stdio::piped())
            .spawn()
            .map_err(|e| format!("Failed to start Spring Boot: {}", e))?;

        *SPRING_BOOT_PROCESS.lock().unwrap() = Some(child);

        // Wait for backend to be ready (check health endpoint)
        println!("Waiting for Spring Boot to be ready...");
        for i in 0..30 {
            thread::sleep(Duration::from_secs(1));
            if is_backend_ready() {
                println!("Spring Boot backend is ready!");
                return Ok(());
            }
            if i % 5 == 0 {
                println!("Still waiting... ({}/30 seconds)", i);
            }
        }
        return Err("Backend did not start within 30 seconds".to_string());
    }

    #[cfg(not(debug_assertions))]
    {
        // Production mode: Use embedded JRE
        start_spring_boot_production()?;
    }

    Ok(())
}

/// Check if backend is ready
fn is_backend_ready() -> bool {
    if let Ok(response) =
reqwest::blocking::get("http://localhost:8080/api/folders") {
        response.status().is_success()
    } else {
        false
```

```
    }
}

/// Start Spring Boot in production mode with embedded JRE
#[cfg(not(debug_assertions))]
fn start_spring_boot_production() -> Result<(), String> {
    use std::env;

    // Determine JRE path based on platform
    let jre_path = if cfg!(target_os = "macos") {
        if cfg!(target_arch = "aarch64") {
            "resources/jre/macos-aarch64/bin/java"
        } else {
            "resources/jre/macos-x64/bin/java"
        }
    } else if cfg!(target_os = "windows") {
        // # Windows configuration - skip for macOS-only development
        // if cfg!(target_arch = "x86_64") {
        //     "resources\\jre\\windows-x64\\bin\\java.exe"
        // } else {
        //     "resources\\jre\\windows-aarch64\\bin\\java.exe"
        // }
        return Err("Windows support not yet implemented".to_string());
    } else {
        return Err("Unsupported platform".to_string());
    };

    let jar_path = "resources/backend.jar";

    let child = Command::new(jre_path)
        .arg("-jar")
        .arg(jar_path)
        .stdout(std::process::Stdio::piped())
        .stderr(std::process::Stdio::piped())
        .spawn()
        .map_err(|e| format!("Failed to start embedded backend: {}", e))?;

    *SPRING_BOOT_PROCESS.lock().unwrap() = Some(child);

    // Wait for startup
    println!("Waiting for Spring Boot to be ready...");
    for _ in 0..30 {
        thread::sleep(Duration::from_secs(1));
        if is_backend_ready() {
            println!("Spring Boot backend is ready!");
            return Ok(());
        }
    }

    Err("Backend did not start within 30 seconds".to_string())
}

/// Shutdown the Spring Boot backend
pub fn shutdown_spring_boot_backend() {
```

```
    if let Some(mut child) = SPRING_BOOT_PROCESS.lock().unwrap().take() {
        println!("Shutting down Spring Boot backend...");
        let _ = child.kill();
        let _ = child.wait();
    }
}
```

Step 4: Update Tauri Commands

Replace your existing Tauri commands with HTTP-based versions:

```
use http_client::BackendClient;

#[tauri::command]
async fn get_all_folders() -> Result<Vec<crate::Folder>, String> {
    let client = BackendClient::new();
    let folders = client.get_all_folders()?;

    Ok(folders
        .into_iter()
        .map(|f| crate::Folder {
            id: f.id,
            name: f.name,
            ordering: f.ordering,
        })
        .collect())
}

#[tauri::command]
async fn create_folder(name: String) -> Result<crate::Folder, String> {
    let client = BackendClient::new();
    let folder = client.create_folder(&name)?;

    Ok(crate::Folder {
        id: folder.id,
        name: folder.name,
        ordering: folder.ordering,
    })
}

#[tauri::command]
async fn update_folder(id: i32, name: String) -> Result<crate::Folder,
String> {
    let client = BackendClient::new();

    // Get current folder to preserve ordering
    let current = client.get_all_folders()?
        .into_iter()
        .find(|f| f.id == id)
        .ok_or("Folder not found")?;
```

```
    let folder = client.update_folder(id, &name, current.ordering)?;

    Ok(crate::Folder {
        id: folder.id,
        name: folder.name,
        ordering: folder.ordering,
    })
}

#[tauri::command]
async fn delete_folder(id: i32) -> Result<(), String> {
    let client = BackendClient::new();
    client.delete_folder(id)
}

// Similar for scripts and app state...
```

Step 5: Initialize on App Start

Update your `run()` function:

```
#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
    // Start Spring Boot backend first
    if let Err(e) = start_spring_boot_backend() {
        eprintln!("Failed to start Spring Boot backend: {}", e);
        eprintln!("The application will now exit.");
        std::process::exit(1);
    }

    tauri::Builder::default()
        .plugin(tauri_plugin_opener::init())
        .invoke_handler(tauri::generate_handler![
            get_all_folders,
            create_folder,
            update_folder,
            delete_folder,
            // ... other commands
        ])
        .setup(|app| {
            // ... existing setup code
            Ok(())
        })
        .on_window_event(|window, event| {
            if let tauri::WindowEvent::CloseRequested { .. } = event {
                // Shutdown Spring Boot when app closes
                shutdown_spring_boot_backend();
            }
        })
        .run(tauri::generate_context!())
```

```
        .expect("error while running tauri application");
}
```

Step 6: Add Chrono Dependency for Timestamps

Add to `Cargo.toml`:

```
[dependencies]
chrono = "0.4"
```

Testing the Integration

Test 1: Start in Development Mode

```
# Terminal 1: Manually start Spring Boot (if not auto-starting)
cd backend-spring
./gradlew bootRun

# Terminal 2: Start Tauri app
cd ..
npm run tauri dev
```

Test 2: Verify API Calls

Add logging to see HTTP requests:

```
// In http_client/mod.rs
pub fn get_all_folders(&self) -> Result<Vec<Folder>, String> {
    let url = format!("{}/folders", BASE_URL);
    println!("GET {}", url);  // Add this
    self.client
        .get(&url)
        .send()
        // ... rest of code
}
```

Test 3: Check Spring Boot Logs

In the Spring Boot terminal, you should see:

```
2024-10-30 15:30:45.123  INFO : GET /api/folders
2024-10-30 15:30:45.456  INFO : POST /api/folders
```

Troubleshooting

Backend not starting

Check:

1. Is Java/JDK installed? `java -version`
2. Is port 8080 free? `lsof -i:8080`
3. Check Rust logs for error messages

Connection refused errors

Issue: Tauri tries to connect before Spring Boot is ready

Solution: Increase wait time in `start_spring_boot_backend()`

```
for i in 0..60 {  // Increase from 30 to 60 seconds
    thread::sleep(Duration::from_secs(1));
    // ...
}
```

CORS errors (if using web frontend separately)

Add CORS config to Spring Boot:

Create `backend-spring/src/main/kotlin/com/scriptmanager/config/CorsConfig.kt`:

```
package com.scriptmanager.config

import org.springframework.context.annotation.Configuration
import org.springframework.web.servlet.config.annotation.CorsRegistry
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer

@Configuration
class CorsConfig : WebMvcConfigurer {
    override fun addCorsMappings(registry: CorsRegistry) {
        registry.addMapping("/**")
            .allowedOrigins("http://localhost:1420")  // Tauri dev server
            .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
    }
}
```

Migration Path

You can migrate gradually:

1. **Phase 1**: Keep both Prisma and Spring Boot running
    - Prisma for reads (fast, local)

    - Spring Boot for writes (ensures validation)

2. **Phase 2**: Switch critical operations to Spring Boot
    - Move create/update/delete to Spring Boot
    - Keep reads on Prisma
3. **Phase 3**: Full migration
    - All operations through Spring Boot
    - Remove Prisma dependency

Performance Considerations

HTTP Overhead: Each call adds ~1-5ms latency

Solutions:

- Batch operations where possible
- Cache frequently accessed data
- Use WebSockets for real-time updates (advanced)

Next Steps

1. Implement all CRUD operations
2. Add error handling and retries
3. Add health check endpoint
4. Set up connection pooling
5. Add request/response logging
6. Implement graceful shutdown

See 3_COMPLETE_SETUP_GUIDE.md for complete setup instructions.