Spring Boot Backend Setup Guide

This guide provides complete instructions for setting up and integrating a Kotlin Spring Boot backend with your Tauri Rust application.

Table of Contents

Overview

The architecture separates concerns as follows:

- **Rust Backend**: Handles Tauri commands, UI state, and launches the Spring Boot backend
- **Spring Boot Backend**: Manages database operations, business logic, and provides REST APIs
- **SQLite Database**: Shared database accessed by Spring Boot (Rust will communicate via HTTP)

Why This Architecture?

- **Separation of Concerns**: Database logic in Spring Boot (Java ecosystem)
- **Maintainability**: JPA entities and Flyway migrations are easier to manage than Prisma
- **Flexibility**: REST API can be consumed by other clients beyond the Tauri app

---

Project Structure

```
.
├── backend-spring/                # Spring Boot project
│   ├── src/
│   │   ├── main/
│   │   │   ├── kotlin/com/scriptmanager/
│   │   │   │   ├── entity/        # JPA entities
│   │   │   │   │   ├── ApplicationState.kt
│   │   │   │   │   ├── ScriptsFolder.kt
│   │   │   │   │   ├── ShellScript.kt
│   │   │   │   │   └── RelScriptsFolderShellScript.kt
│   │   │   │   ├── repository/    # Spring Data JPA repositories
│   │   │   │   ├── controller/    # REST controllers
```

```
│   │   │   │   └── Application.kt # Main application
│   │   │   └── resources/
│   │   │       ├── application.yml
│   │   │       └── db/migration/  # Flyway migrations
│   │   │           └── V1__Initial_schema.sql
│   │   └── test/                  # Tests
│   ├── build.gradle.kts           # Gradle build configuration
│   ├── settings.gradle.kts
│   ├── FLYWAY_WORKFLOW.md         # Detailed Flyway instructions
│   └── gradlew                    # Gradle wrapper
├── src-tauri/                     # Rust Tauri backend
│   ├── database.db                # Shared SQLite database
│   └── ...
└── src/                           # Frontend React app
```

Prerequisites

Required Software

1. **Java Development Kit (JDK) 17 or higher**

   ```
   # Check Java version
   java -version

   # Install via Homebrew (macOS)
   brew install openjdk@17
   ```

2. **Gradle** (optional - we use Gradle wrapper)

   ```
   # Check Gradle version
   gradle --version
   ```

3. **Rust and Cargo** (already installed for Tauri)

4. **Node.js and npm/yarn** (already installed for Tauri)

Optional Tools

- **IntelliJ IDEA** (recommended for Kotlin development)
- **Postman** or **curl** (for API testing)
- **SQLite Browser** (for database inspection)

Initial Setup

Step 1: Verify Project Structure

Navigate to the backend-spring directory:

```
cd backend-spring
```

Verify all files are present:

```
ls -la
# Should show: build.gradle.kts, settings.gradle.kts, src/, gradlew, etc.
```

Step 2: Test Gradle Build

```
# Make gradlew executable (if needed)
chmod +x gradlew

# Build the project
./gradlew build

# This will:
# - Download dependencies
# - Compile Kotlin code
# - Run tests
# - Create JAR file in build/libs/
```

Expected output:

```
BUILD SUCCESSFUL in 30s
```

Step 3: Run the Spring Boot Application

```
./gradlew bootRun
```

The application should start on `http://localhost:8080`

You should see logs like:

```
2024-10-30 ... Started Application in 3.5 seconds
Flyway: Successfully validated 1 migration(s)
```

Step 4: Test the API

In another terminal:

```
# Get all folders
curl http://localhost:8080/api/folders

# Get application state
curl http://localhost:8080/api/app-state
```

Database Configuration

SQLite Connection

The Spring Boot app connects to the existing SQLite database created by Tauri:

File: `backend-spring/src/main/resources/application.yml`

```
spring:
  datasource:
    url: jdbc:sqlite:../src-tauri/database.db # Shared database
    driver-class-name: org.sqlite.JDBC

  jpa:
    database-platform: org.hibernate.community.dialect.SQLiteDialect
    hibernate:
      ddl-auto: validate # IMPORTANT: Only validate, never auto-update
```

Key Configuration Points

1. **Database Location**: Points to `../src-tauri/database.db` (relative path)
2. **Hibernate DDL Mode**: Set to `validate` (never `create`, `update`, or `create-drop`)
3. **Flyway Enabled**: Handles all schema changes

Flyway Migration Workflow

Understanding the Workflow

CRITICAL: JPA entities do NOT automatically update the database schema. All schema changes must go through Flyway migrations.

Step-by-Step: Making Schema Changes

Example: Adding a `description` field to `ShellScript`

Step 1: Modify the JPA Entity

Edit: `backend-spring/src/main/kotlin/com/scriptmanager/entity/ShellScript.kt`

```
@Entity
@Table(name = "shell_script")
data class ShellScript(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Int? = null,

    val name: String = "",
    val command: String = "",
    val ordering: Int = 0,

    // NEW FIELD
    val description: String? = null,

    @Column(name = "created_at")
    val createdAt: Double = 0.0,

    @Column(name = "created_at_hk")
    val createdAtHk: String = ""
)
```

Step 2: Create a Flyway Migration

Create: `backend-spring/src/main/resources/db/migration/V2__Add_description_to_shell_script.sql`

```
-- Add description column to shell_script table
ALTER TABLE shell_script ADD COLUMN description TEXT;
```

Naming Convention:

- `V{version}__{description}.sql`
- `V2__Add_description.sql` ✓
- `V3__Create_user_table.sql` ✓
- `v2_add.sql` ✗ (wrong format)

Step 3: Apply the Migration

```
# Stop the Spring Boot app if running (Ctrl+C)

# Restart - Flyway will auto-apply pending migrations
./gradlew bootRun
```

You'll see:

```
Flyway: Migrating schema to version "2 - Add description to shell script"
Flyway: Successfully applied 1 migration
```

Step 4: Verify

```
# Check the schema was updated
sqlite3 ../src-tauri/database.db "PRAGMA table_info(shell_script);"
```

You should see the new `description` column.

Step 5: Commit Both Files

```
git add backend-
spring/src/main/kotlin/com/scriptmanager/entity/ShellScript.kt
git add backend-
spring/src/main/resources/db/migration/V2__Add_description_to_shell_script
.sql
git commit -m "Add description field to shell_script"
```

More Migration Examples

See `5_FLYWAY_WORKFLOW.md` for detailed examples including:

- Adding new tables
- Modifying columns (SQLite limitations)
- Creating indexes
- Data migrations
- Rollback strategies

Development Mode Integration

Option 1: Manual Launch (Current Setup)

1. **Terminal 1**: Run Spring Boot backend

   ```
   cd backend-spring
   ./gradlew bootRun
   ```

2. **Terminal 2**: Run Tauri app

   ```
   cd ..
   npm run tauri dev
   ```

3. Update Rust code to call Spring Boot APIs instead of direct database access

Option 2: Auto-Launch from Rust (Recommended)

Modify `src-tauri/src/lib.rs` to launch Spring Boot on startup:

Add to Cargo.toml:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
reqwest = { version = "0.11", features = ["json"] }
```

Add to lib.rs:

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

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

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

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

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

        // Wait for backend to be ready
        std::thread::sleep(std::time::Duration::from_secs(10));
    }

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

    Ok(())
}

// Call this when app starts
// In your setup function:
```

```
pub fn run() {
    // Start Spring Boot first
    if let Err(e) = start_spring_boot_backend() {
        eprintln!("Failed to start backend: {}", e);
        return;
    }

    // ... rest of your Tauri setup
}
```

Production Mode (Embedded JRE)

For production, embed a JRE so users don't need Java installed.

Step 1: Download JRE

Download **JRE 17** (smaller than JDK) for your target platforms:

For macOS (ARM64):

```
# Create directory for embedded JRE
mkdir -p src-tauri/resources/jre

# Download and extract JRE (example - use actual download link)
# Option 1: Temurin (recommended)
# Download from: https://adoptium.net/temurin/releases/

# Option 2: Use jlink to create minimal JRE
jlink --add-modules
java.base,java.sql,java.naming,java.desktop,java.management,java.instrumen
t \
      --output src-tauri/resources/jre/macos-aarch64 \
      --strip-debug \
      --no-man-pages \
      --no-header-files \
      --compress=2
```

For Windows (x64):

```
# Windows JRE - skip for now if on macOS
# mkdir -p src-tauri/resources/jre/windows-x64
```

Step 2: Build Fat JAR

```
cd backend-spring
./gradlew bootJar
```

```
# Output: build/libs/script-manager-backend-0.0.1-SNAPSHOT.jar
```

Copy to resources:

```
mkdir -p ../src-tauri/resources
cp build/libs/script-manager-backend-0.0.1-SNAPSHOT.jar \
   ../src-tauri/resources/backend.jar
```

Step 3: Configure Tauri to Bundle Resources

Edit `src-tauri/tauri.conf.json`:

```
{
  "bundle": {
    "resources": ["resources/backend.jar", "resources/jre/**/*"]
  }
}
```

Step 4: Update Rust Production Code

```
#[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 - commented out for macOS-only
development
        // "resources\\jre\\windows-x64\\bin\\java.exe"
        return Err("Windows not yet configured".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)
        .spawn()
```

```
            .map_err(|e| format!("Failed to start embedded backend: {}", e))?;

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

        // Wait for startup
        std::thread::sleep(std::time::Duration::from_secs(5));

        Ok(())
    }
```

Step 5: Test Production Build

```
# Build Tauri in production mode
npm run tauri build

# Test the built app
./src-tauri/target/release/bundle/macos/YourApp.app/Contents/MacOS/YourApp
```

Platform-Specific Notes

macOS:

- Use ARM64 (aarch64) JRE for Apple Silicon
- Use x64 JRE for Intel Macs
- May need to sign the JRE for distribution

Windows (commented out in code):

```
// # Windows-specific code
// # Uncomment and implement when ready to support Windows:
// #[cfg(target_os = "windows")]
// fn get_windows_jre_path() -> &'static str {
//     if cfg!(target_arch = "x86_64") {
//         "resources\\jre\\windows-x64\\bin\\java.exe"
//     } else {
//         "resources\\jre\\windows-aarch64\\bin\\java.exe"
//     }
// }
```

API Endpoints

Folders

```
# Get all folders
GET /api/folders
```

```
# Get folder by ID
GET /api/folders/{id}

# Create folder
POST /api/folders
Content-Type: application/json
{
  "name": "My Scripts",
  "ordering": 0,
  "createdAt": 1698765432000.0,
  "createdAtHk": "2024-10-30 15:30:32"
}

# Update folder
PUT /api/folders/{id}
Content-Type: application/json
{
  "name": "Updated Name",
  "ordering": 1,
  "createdAt": 1698765432000.0,
  "createdAtHk": "2024-10-30 15:30:32"
}

# Delete folder
DELETE /api/folders/{id}
```

Scripts

```
# Get all scripts
GET /api/scripts

# Get script by ID
GET /api/scripts/{id}

# Create script
POST /api/scripts
Content-Type: application/json
{
  "name": "Build Project",
  "command": "npm run build",
  "ordering": 0,
  "createdAt": 1698765432000.0,
  "createdAtHk": "2024-10-30 15:30:32"
}

# Update script
PUT /api/scripts/{id}
Content-Type: application/json
{
  "name": "Updated Name",
```

```
  "command": "npm run build:prod",
  "ordering": 1,
  "createdAt": 1698765432000.0,
  "createdAtHk": "2024-10-30 15:30:32"
}

# Delete script
DELETE /api/scripts/{id}
```

Application State

```
# Get application state
GET /api/app-state

# Update application state
PUT /api/app-state
Content-Type: application/json
{
  "id": 1,
  "lastOpenedFolderId": 5,
  "darkMode": true,
  "createdAt": 1698765432000.0,
  "createdAtHk": "2024-10-30 15:30:32"
}
```

Testing

Test the Spring Boot Backend

```
cd backend-spring

# Run all tests
./gradlew test

# Run with verbose output
./gradlew test --info

# Run specific test class
./gradlew test --tests "FolderControllerTest"
```

Integration Testing

Create a test file: `backend-spring/src/test/kotlin/com/scriptmanager/FolderIntegrationTest.kt`

```
package com.scriptmanager

import org.junit.jupiter.api.Test
import org.springframework.beans.factory.annotation.Autowired
import
org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockM
vc
import org.springframework.boot.test.context.SpringBootTest
import org.springframework.test.web.servlet.MockMvc
import
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*
import org.springframework.test.web.servlet.result.MockMvcResultMatchers.*

@SpringBootTest
@AutoConfigureMockMvc
class FolderIntegrationTest {

    @Autowired
    private lateinit var mockMvc: MockMvc

    @Test
    fun `should get all folders`() {
        mockMvc.perform(get("/api/folders"))
            .andExpect(status().isOk)
            .andExpect(content().contentType("application/json"))
    }
}
```

Manual API Testing

```
# Test creating a folder
curl -X POST http://localhost:8080/api/folders \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Test Folder",
    "ordering": 0,
    "createdAt": 1698765432000.0,
    "createdAtHk": "2024-10-30 15:30:32"
  }'

# Test getting all folders
curl http://localhost:8080/api/folders

# Test updating a folder
curl -X PUT http://localhost:8080/api/folders/1 \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Updated Folder",
    "ordering": 0,
    "createdAt": 1698765432000.0,
```

```
    "createdAtHk": "2024-10-30 15:30:32"
  }'
```

Development Workflow Summary

Daily Development

1. **Start Spring Boot Backend**:

```
cd backend-spring
./gradlew bootRun
```

2. **Start Tauri App** (in another terminal):

```
cd ..
npm run tauri dev
```

3. **Make changes**:
   - Edit Kotlin code
   - Spring Boot auto-reloads with DevTools
   - Edit React/Rust code
   - Tauri hot-reloads

Making Database Changes

1. Edit JPA entity
2. Create Flyway migration SQL file
3. Restart Spring Boot (applies migration automatically)
4. Test changes
5. Commit both entity + migration file

Building for Production

1. **Build Spring Boot JAR**:

```
cd backend-spring
./gradlew bootJar
```

2. **Copy to Tauri resources**:

```
cp build/libs/*.jar ../src-tauri/resources/backend.jar
```

3. **Build Tauri app**:

```
cd ..
npm run tauri build
```

Troubleshooting

Spring Boot won't start

Error: "Address already in use"

```
# Kill process on port 8080
lsof -ti:8080 | xargs kill -9
```

Error: "Database is locked"

- Close any SQLite browser connections
- Stop any running Spring Boot instances

Flyway migration failed

Error: "Migration checksum mismatch"

- Don't modify existing migration files
- Create a new migration to fix issues

Error: "Validation failed"

- Ensure migrations are numbered sequentially (V1, V2, V3...)
- Check for duplicate version numbers

JPA entity doesn't match schema

Error: "Schema validation failed"

- Check entity annotations match database columns
- Verify data types (TEXT vs String, INTEGER vs Int)
- Create missing migration

Embedded JRE issues

Error: "JRE not found"

- Verify JRE is copied to `src-tauri/resources/jre/`
- Check Tauri bundle configuration includes JRE

Error: "Permission denied"

```
# Make JRE executable
chmod +x src-tauri/resources/jre/*/bin/java
```

Next Steps

1. ✅ Setup complete - Spring Boot backend running
2. ✅ Flyway migrations configured
3. ✅ JPA entities created
4. ⏳ **TODO**: Update Rust code to call Spring Boot APIs instead of direct DB access
5. ⏳ **TODO**: Create embedded JRE for production builds
6. ⏳ **TODO**: Add comprehensive tests

Additional Resources

- Spring Boot Documentation
- Kotlin Language Guide
- Flyway Documentation
- Spring Data JPA
- 5_FLYWAY_WORKFLOW.md - Detailed Flyway guide

Support

For issues or questions:

1. Check this documentation
2. Review `5_FLYWAY_WORKFLOW.md`
3. Check Spring Boot logs in terminal
4. Inspect database with SQLite browser

Last Updated: October 30, 2025 **Project**: Shell Script Manager - Spring Boot Backend