# Spring Boot Backend - Complete Setup Overview

## **6** What Has Been Created

I've set up a complete **Kotlin Spring Boot backend** for your Tauri application with the following features:

- Spring Boot 3.2.0 with Kotlin 1.9.20
- JPA entities matching your existing database schema
- SQLite database integration (shares database with Tauri)
- Flyway migrations for schema management
- REST API for folders, scripts, and app state
- Gradle build system with wrapper included
- Development and production modes configured
- Comprehensive documentation for all workflows

# Project Structure



# Documentation Guide

I've created 4 comprehensive documentation files to help you:

#### 1. **3\_COMPLETE\_SETUP\_GUIDE.md** (Main Guide)

- Location: Project root
- Purpose: Complete end-to-end setup instructions

#### • Contents:

- Prerequisites and installation
- Initial setup steps
- o Database configuration
- Flyway migration workflow
- Development and production integration
- o API endpoint documentation
- Testing instructions
- o Troubleshooting guide

## 2. 4\_QUICK\_START.md

• Purpose: Get running in 5 minutes

- Contents:
  - TL;DR commands
  - Quick API tests
  - o Common commands reference
  - o API endpoint table

## 3. 5\_FLYWAY\_WORKFLOW.md

- Purpose: Detailed database migration workflow
- Contents:
  - How to change JPA entities
  - How to create migration SQL files
  - o Migration examples (add column, create table, modify column)
  - Best practices
  - Troubleshooting migrations

## 4. 6\_RUST\_INTEGRATION.md

- Purpose: Integrate Spring Boot with Rust/Tauri
- Contents:
  - Complete Rust code examples
  - HTTP client implementation
  - Auto-launching Spring Boot from Rust
  - Development and production modes
  - Testing and troubleshooting



#### Step 1: Test the Spring Boot Backend

# Navigate to backend
cd backend-spring

# Build the project

```
./gradlew build

# Run the application
./gradlew bootRun
```

The server will start at http://localhost:8080

## Step 2: 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
```

#### Step 3: Read the Documentation

Start with: 3\_COMPLETE\_SETUP\_GUIDE.md



## 1. JPA Entities Do NOT Auto-Update Schema

This is **CRITICAL** to understand:

```
# application.yml
spring:
    jpa:
    hibernate:
    ddl-auto: validate # ← Only validates, never creates/updates
```

#### Why?

- We use Flyway for all schema changes
- Ensures version control of database changes
- Prevents accidental schema modifications
- · Allows safe rollbacks

#### 2. Schema Change Workflow

Every schema change follows these steps:

- 1. Modify JPA entity (add/remove/change field)
- 2. Create Flyway migration (SQL file: V2\_\_Description.sql)

- 3. Restart app (Flyway auto-applies migration)
- 4. **Test and commit** (both entity + migration)

**Example**: Add a description field to ShellScript

```
// 1. Edit entity
@Entity
data class ShellScript(
    // ... existing fields ...
    val description: String? = null // NEW
)
```

```
-- 2. Create: V2__Add_description_to_shell_script.sql
ALTER TABLE shell_script ADD COLUMN description TEXT;
```

```
# 3. Restart (auto-applies migration)
./gradlew bootRun
```

See 5\_FLYWAY\_WORKFLOW.md for detailed examples.

## 3. Development vs Production

#### **Development Mode:**

- Spring Boot runs via \_/gradlew bootRun
- Can be auto-launched from Rust
- Uses local Java installation

#### **Production Mode:**

- Spring Boot runs as embedded JAR
- Uses bundled JRE (no Java installation needed)
- Embedded in Tauri app bundle

See 3\_COMPLETE\_SETUP\_GUIDE.md sections 7 & 8.

## 4. Rust Integration

Your Rust backend will:

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

See 6\_RUST\_INTEGRATION.md for complete code.

## Database Schema

The Spring Boot backend connects to your existing SQLite database:

#### Tables:

- application\_state App configuration and state
- scripts\_folder Folder definitions
- shell\_script Script definitions
- rel\_scriptsfolder\_shellscript Folder-script relationships

JPA Entities (in backend-spring/src/main/kotlin/com/scriptmanager/entity/):

- ApplicationState.kt
- ScriptsFolder.kt
- ShellScript.kt
- RelScriptsFolderShellScript.kt

# REST API Endpoints

#### **Folders**

- GET /api/folders List all folders
- GET /api/folders/{id} Get specific folder
- POST /api/folders Create folder
- PUT /api/folders/{id} Update folder
- DELETE /api/folders/{id} Delete folder

#### **Scripts**

- GET /api/scripts List all scripts
- GET /api/scripts/{id} Get specific script
- POST /api/scripts Create script
- PUT /api/scripts/{id} Update script

• DELETE /api/scripts/{id} - Delete script

#### **Application State**

- GET /api/app-state Get app state
- PUT /api/app-state Update app state

# X Common Tasks

#### Task 1: Run the Backend

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

## Task 2: Build Production JAR

```
cd backend-spring
./gradlew bootJar
# Output: build/libs/script-manager-backend-0.0.1-SNAPSHOT.jar
```

## Task 3: Add a New Field to Entity

**Example**: Add description to ShellScript

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

```
val description: String? = null
```

2. Create backend-spring/src/main/resources/db/migration/V2\_Add\_description.sql:

```
ALTER TABLE shell_script ADD COLUMN description TEXT;
```

3. Restart app:

```
./gradlew bootRun
```

#### Task 4: Integrate with Rust

Follow the complete example in 6\_RUST\_INTEGRATION.md

Summary:

- 1. Add dependencies: regwest, tokio, chrono
- 2. Create HTTP client module
- 3. Launch Spring Boot on app startup
- 4. Replace Prisma calls with HTTP calls

#### Task 5: Prepare for Production

- 1. **Download JRE 17** for your platform(s)
- 2. Place in src-tauri/resources/jre/
- 3. Build backend JAR: ./gradlew bootJar
- 4. Copy to src-tauri/resources/backend.jar
- 5. Update Rust to launch embedded JAR
- 6. Build Tauri: npm run tauri build

See 3\_COMPLETE\_SETUP\_GUIDE.md section 8 for details.

# Learning Path

## Day 1: Setup and Testing

- 1. Read 4\_QUICK\_START.md
- 2. Run ./gradlew bootRun
- 3. Test API endpoints with curl
- 4. Explore the code in IntelliJ IDEA or VS Code

#### Day 2: Understanding Flyway

- 1. Read 5\_FLYWAY\_WORKFLOW.md
- 2. Practice: Add a simple field to an entity
- 3. Create your first migration
- 4. Apply and verify

#### Day 3: Rust Integration

- 1. Read 6\_RUST\_INTEGRATION.md
- 2. Add HTTP client code to Rust
- 3. Test launching Spring Boot from Rust
- 4. Convert one Prisma operation to HTTP call

#### Day 4: Full Integration

- 1. Convert all operations to HTTP calls
- 2. Test development mode
- 3. Handle errors and edge cases
- 4. Add logging and monitoring

#### Day 5: Production Preparation

1. Read 3\_COMPLETE\_SETUP\_GUIDE.md section 8

- 2. Download and bundle JRE
- 3. Create production build
- 4. Test embedded deployment

# Troubleshooting

Problem: Port 8080 already in use

```
lsof -ti:8080 | xargs <mark>kill</mark> -9
```

Problem: Database is locked

- · Close SQLite browser
- Stop all Spring Boot instances
- Check for database.db-journal files

Problem: Migration failed

- Don't modify existing migration files
- Check SQL syntax (SQLite compatibility)
- Ensure sequential version numbers

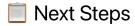
Problem: Build failed

```
# Clean and rebuild
cd backend-spring
./gradlew clean build --no-daemon
```

Problem: Rust can't connect to backend

- Check backend is running: curl http://localhost:8080/api/folders
- Increase wait time in Rust startup code
- · Check firewall/security settings

For more troubleshooting, see 3\_COMPLETE\_SETUP\_GUIDE.md section 10.



Immediate (Today)

- Run \_/gradlew bootRun and verify it works
- Test API endpoints with curl
- Read 3\_COMPLETE\_SETUP\_GUIDE.md

Short Term (This Week)

• Follow 6_RUST_INTEGRATION.md	
Add HTTP client to Rust	
<ul> <li>Auto-launch Spring Boot from Rust</li> </ul>	
<ul> <li>Convert one operation to HTTP (e.g., get_all_folders)</li> </ul>	
Medium Term (Next Week)	
Convert all operations to HTTP calls	
Remove Prisma dependency (optional)	
Add error handling and retries	
Write integration tests	
Long Term (Production)	
<ul> <li>Download and bundle JRE for Mac ARM64</li> </ul>	
<ul> <li>■ Bundle JRE for Mac x64 (optional)</li> </ul>	
<ul> <li>Bundle JRE for Windows (optional)</li> </ul>	
Test production builds	

# 💡 Key Benefits of This Architecture

#### 1. Separation of Concerns

Deploy to users

- o Rust: UI, system integration, Tauri commands
- o Spring Boot: Database, business logic, validation

#### 2. Maintainability

- o JPA entities are easier to work with than Prisma
- Flyway provides version control for database
- Spring Boot ecosystem is mature and well-documented

## 3. Flexibility

- REST API can be used by other clients
- Can deploy backend separately if needed
- Easy to add authentication, caching, etc.

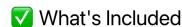
#### 4. Developer Experience

- Hot reload with Spring DevTools
- Great IDE support (IntelliJ IDEA)
- Rich ecosystem of libraries and tools

## Documentation Index

Document	Purpose	Read When	
----------	---------	-----------	--

Document	Purpose	Read When
2_OVERVIEW.md	Overview (this file)	Start here
3_COMPLETE_SETUP_GUIDE.md	Complete setup guide	Setting up for first time
4_QUICK_START.md	Quick reference	Need fast commands
5_FLYWAY_WORKFLOW.md	Database migrations	Changing schema
6_RUST_INTEGRATION.md	Rust integration	Connecting Rust to Spring
backend-spring/README.md	Project README	Understanding structure



#### Code

- Complete Spring Boot project structure
- 4 JPA entities matching your schema
- **V** 4 Spring Data repositories
- **☑** 3 REST controllers with full CRUD
- V Flyway initial migration
- **V** Gradle build configuration
- Application configuration (YAML)

#### Documentation

- ✓ Complete setup guide (31 sections)
- Quick start guide (5 minutes)
- V Flyway workflow guide (with examples)
- **V** Rust integration guide (with full code)
- V Project README
- **V** This overview

#### Configuration

- Gradle wrapper (no Gradle install needed)
- SQLite dialect configured
- V Flyway enabled and configured
- V Development mode ready
- V Production mode documented

# You're All Set!

Everything is configured and ready to use. Follow these steps:

- 1. Read 3\_COMPLETE\_SETUP\_GUIDE.md
- 2. Run cd backend-spring && ./gradlew bootRun
- 3. Test curl http://localhost:8080/api/folders

4. Integrate with Rust using 6\_RUST\_INTEGRATION.md

# **Support**

All questions should be answered in the documentation. If not:

- 1. Check the relevant documentation file
- 2. Look at code comments
- 3. Review Spring Boot logs
- 4. Check database with SQLite browser

Created: October 30, 2025

**Technology**: Spring Boot 3.2.0 + Kotlin 1.9.20 + SQLite + Flyway

**Status**: **✓** Complete and ready for development

Happy coding! 🚀