

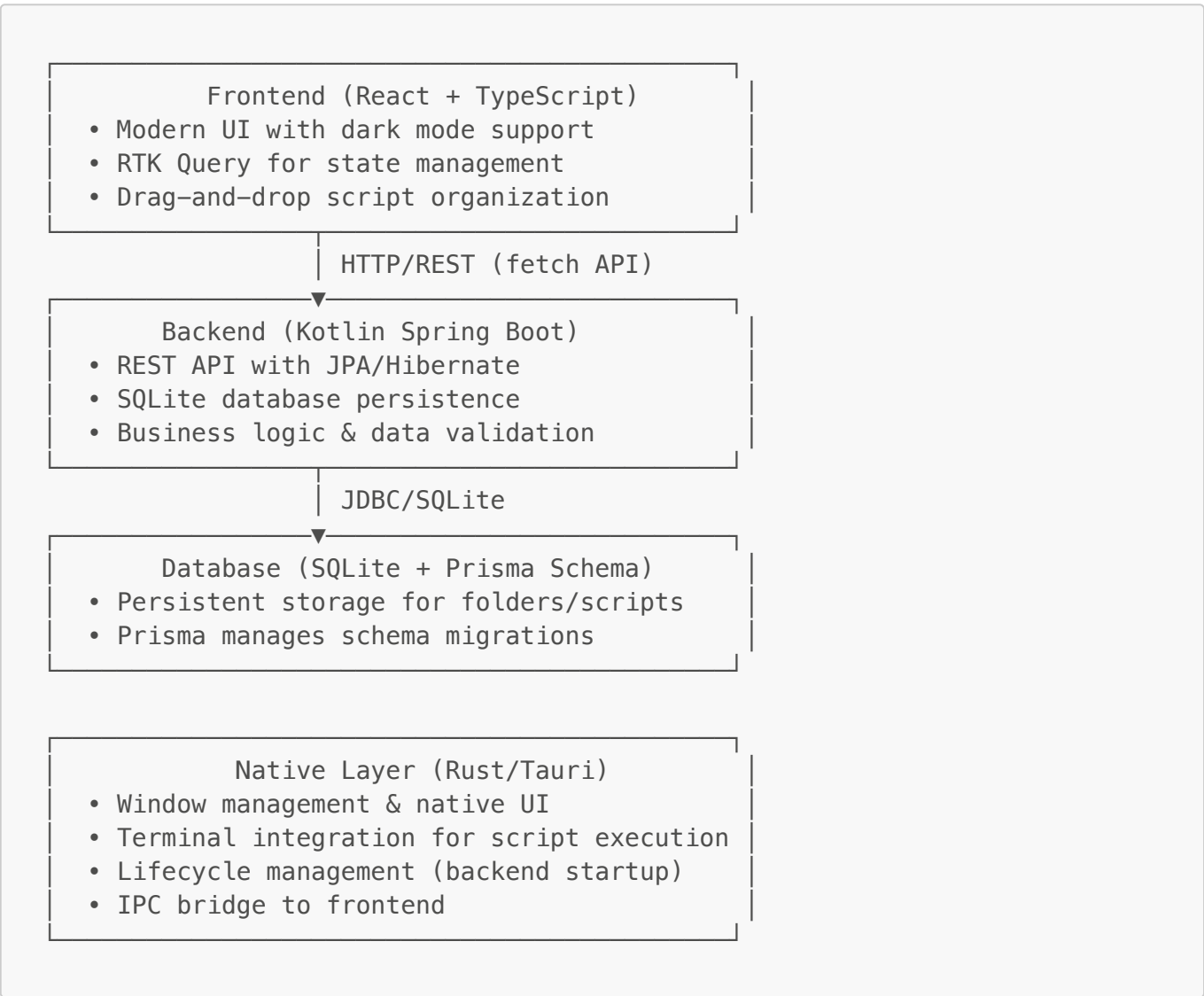
Shell Script Manager - Complete Project Summary

Project Overview

A modern desktop application for macOS that helps users organize and execute shell scripts through an intuitive GUI. Built with a hybrid architecture combining Rust (Tauri), React (TypeScript), and Kotlin (Spring Boot).

Architecture

Three-Tier Architecture:



Tech Stack

Frontend

- **Framework:** React 18 + TypeScript
- **UI Components:** Custom components with Tailwind CSS + shadcn/ui
- **State Management:** Redux Toolkit + RTK Query
- **Drag & Drop:** @dnd-kit for reordering

- **Build Tool:** Vite

Backend

- **Language:** Kotlin
- **Framework:** Spring Boot 3.x
- **Database:** SQLite with Hibernate ORM
- **Build System:** Gradle (Kotlin DSL)
- **API:** REST with automatic JSON serialization

Native/Desktop Layer

- **Framework:** Tauri v2
- **Language:** Rust
- **Database Schema:** Prisma (schema definition & migrations)
- **Build Tool:** Cargo

Project Structure

```
shell-script-manager-tauri/
├── src/                                # React frontend
│   ├── app-component/                 # Main UI components
│   │   ├── FolderColumn/             # Folder list & management
│   │   └── ScriptsColumn/            # Script list & execution
│   ├── components/                   # Reusable UI components
│   ├── store/                        # Redux store & API slices
│   │   ├── api/                     # RTK Query endpoints
│   │   └── slices/                  # Redux state slices
│   └── hooks/                        # Custom React hooks
├── src-tauri/                         # Rust native layer
│   ├── src/lib.rs                    # Core Tauri application logic
│   ├── prisma/schema.prisma          # Database schema definition
│   └── Cargo.toml                    # Rust dependencies
├── backend-spring/                   # Spring Boot backend
│   ├── src/main/kotlin/
│   │   └── com/scriptmanager/
│   │       ├── controller/           # REST API endpoints
│   │       ├── common/
│   │       │   ├── entity/          # JPA entities
│   │       │   └── dto/             # Data transfer objects
│   │       └── repository/          # Spring Data repositories
│   └── build.gradle.kts              # Gradle build configuration
└── docs/                             # Project documentation
```

Communication Bridges Between Components

1. Frontend ↔ Backend (HTTP/REST via RTK Query)

Frontend Side - RTK Query Base Configuration

```
// src/store/api/baseQuery/httpBaseQuery.ts
export const httpBaseQuery = (): BaseQueryFn<
  HttpQueryArgs,
  unknown,
  HttpQueryError
> => {
  return async ({ url, method = "GET", body, params }, api) => {
    // Get dynamic backend URL from Redux state
    const state = api.getState() as RootState;
    const port = state.config.backendPort;
    const backendUrl = `http://localhost:${port}`;

    // Build URL with query params
    const fullUrl = new URL(`${backendUrl}${url}`);
    if (params) {
      Object.entries(params).forEach(([key, value]) => {
        fullUrl.searchParams.append(key, value);
      });
    }

    // Make fetch request
    const response = await fetch(fullUrl.toString(), {
      method,
      headers: { "Content-Type": "application/json" },
      body: body ? JSON.stringify(body) : undefined,
    });

    if (!response.ok) {
      const errorData = await response.text();
      return { error: { status: response.status, data: errorData } };
    }

    const data = await response.json();
    return { data: data.result };
  };
};
```

Frontend Side - RTK Query API Definition

```
// src/store/api/folderApi.ts
export const folderApi = baseApi.injectEndpoints({
  endpoints: (builder) => ({
    getAllFolders: builder.query<ScriptsFolderDT0[], void>({
      query: () => ({
        url: "/folders",
        method: "GET",
      })
    })
  })
});
```

```

    }),
    providesTags: ["Folder"],
  }),

  createFolder: builder.mutation<ScriptsFolderDT0, CreateFolderRequest>
({
  query: (request) => ({
    url: "/folders",
    method: "POST",
    body: request,
  }),
  invalidatesTags: ["Folder"],
}),
});

```

Frontend Side - Using the API in Components

```

// src/app-component/FolderColumn/FolderColumn.tsx
export default function FolderColumn() {
  const backendPort = useAppSelector((s) => s.config.backendPort);

  // Query only runs when backendPort is available
  const { data: folders, isLoading } =
    folderApi.endpoints.getAllFolders.useQuery(undefined, {
      skip: !backendPort,
      selectFromResult: (result) => ({
        ...result,
        data: result.data ?? [],
      }),
    });

  const [createFolder] = folderApi.endpoints.createFolder.useMutation();

  const handleCreateFolder = async () => {
    await createFolder({ name: newFolderName.trim() });
  };
}

```

Backend Side - Spring Boot REST Controller

```

// backend-
spring/src/main/kotlin/com/scriptmanager/controller/FolderController.kt
@RestController
@RequestMapping("/folders")
class FolderController(
  private val folderRepository: ScriptsFolderRepository
) {

```

```

    @GetMapping
    fun getAllFolders(): ResponseEntity<ApiResponse<List<ScriptsFolder>>>
    {
        val folders = folderRepository.findAllByOrderByOrderIndexAsc()
        return ResponseEntity.ok(ApiResponse.success(folders))
    }

    @PostMapping
    fun createFolder(@RequestBody request: CreateFolderRequest):
    ResponseEntity<ApiResponse<ScriptsFolder>> {
        val maxOrder = folderRepository.findMaxOrderIndex() ?: -1
        val newFolder = ScriptsFolder(
            name = request.name,
            orderIndex = maxOrder + 1
        )
        val saved = folderRepository.save(newFolder)
        return ResponseEntity.ok(ApiResponse.success(saved))
    }
}

```

2. Backend ↔ Database (JPA/Spring Data)

Schema Definition - Prisma (Source of Truth)

```

// src-tauri/prisma/schema.prisma
model ScriptsFolder {
  id          Int      @id @default(autoincrement())
  name        String
  orderIndex  Int       @map("order_index")
  createdAt   DateTime @default(now()) @map("created_at")
  createdAtHk String    @default("0") @map("created_at_hk")

  scripts ShellScript[]

  @@map("scripts_folder")
}

model ShellScript {
  id          Int      @id @default(autoincrement())
  name        String
  command     String
  orderIndex  Int       @map("order_index")
  folderId    Int       @map("folder_id")
  createdAt   DateTime @default(now()) @map("created_at")
  createdAtHk String    @default("0") @map("created_at_hk")

  folder ScriptsFolder @relation(fields: [folderId], references: [id],
onDelete: Cascade)
}

```

```

    @@map("shell_script")
}

```

JPA Entity (Matches Prisma Schema)

```

// backend-
spring/src/main/kotlin/com/scriptmanager/common/entity/ScriptsFolder.kt
@Entity
@Table(name = "scripts_folder")
@DynamicInsert
data class ScriptsFolder(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    val id: Long = 0,

    @Column(name = "name", nullable = false)
    var name: String,

    @Column(name = "order_index", nullable = false)
    var orderIndex: Int,

    @Column(name = "created_at")
    var createdAt: LocalDateTime? = null,

    @Column(name = "created_at_hk")
    var createdAtHk: String? = null,

    @OneToMany(mappedBy = "folder", cascade = [CascadeType.ALL], fetch =
FetchType.LAZY)
    val scripts: MutableList<ShellScript> = mutableListOf()
)

```

Spring Data Repository

```

// backend-
spring/src/main/kotlin/com/scriptmanager/repository/ScriptsFolderRepository.kt
@Repository
interface ScriptsFolderRepository : JpaRepository<ScriptsFolder, Long> {
    fun findAllByOrderByOrderIndexAsc(): List<ScriptsFolder>

    @Query("SELECT MAX(f.orderIndex) FROM ScriptsFolder f")
    fun findMaxOrderIndex(): Int?
}

```

Database Configuration

```
# backend-spring/src/main/resources/application.yml
spring:
  datasource:
    url: jdbc:sqlite:${DB_PATH:/path/to/database.db} # Overridden by Rust
    driver-class-name: org.sqlite.JDBC

  jpa:
    database-platform: org.hibernate.community.dialect.SQLiteDialect
    hibernate:
      ddl-auto: none # Prisma manages schema
    properties:
      hibernate:
        jdbc:
          use_get_generated_keys: false # Required for SQLite
```

3. Rust ↔ Database (Prisma Client)

Database Initialization in Rust

```
// src-tauri/src/lib.rs
pub fn init_db(app_handle: &tauri::AppHandle) -> Result<(), String> {
  // Get database path (dev vs production)
  let db_path = get_database_path(app_handle)?;
  println!("Database location: {}", db_path);

  // Set DATABASE_URL for Prisma
  let database_url = format!("file:{}", db_path);
  std::env::set_var("DATABASE_URL", &database_url);

  // Spawn separate thread to avoid async runtime conflicts
  let database_url_clone = database_url.clone();
  std::thread::spawn(move || {
    std::env::set_var("DATABASE_URL", &database_url_clone);

    let rt = tokio::runtime::Runtime::new().expect("Failed to create runtime");
    rt.block_on(async move {
      let client = prisma::new_client_with_url(&database_url_clone)
        .await
        .expect("Failed to create Prisma client");

      // Sync schema (allows automatic updates in production)
      client
        ._db_push()
        .accept_data_loss()
        .await
        .expect("Failed to sync database schema");

      PRISMA_CLIENT
        .set(client)
    });
  });
}
```

```

        .expect("Failed to set Prisma client");
    });
})
.join()
.map_err(|_| "Failed to initialize database".to_string())?;

Ok(())
}

```

Dynamic Database Path

```

// src-tauri/src/lib.rs
fn get_database_path(app_handle: &tauri::AppHandle) -> Result<String,
String> {
    #[cfg(debug_assertions)]
    {
        // Development: use src-tauri/database.db
        let current_dir = std::env::current_dir()
            .map_err(|e| format!("Failed to get current directory: {}",
e))?.;
        let db_path = current_dir.join("database.db");
        Ok(db_path.to_string_lossy().to_string())
    }

    #[cfg(not(debug_assertions))]
    {
        // Production: use Application Support directory
        let app_name = env!("CARGO_PKG_NAME");
        let home_dir = std::env::var("HOME")
            .map_err(|e| format!("Failed to get HOME directory: {}", e))?.;

        #[cfg(target_os = "macos")]
        let app_support_dir = std::path::PathBuf::from(&home_dir)
            .join("Library")
            .join("Application Support")
            .join(app_name);

        // Create directory if it doesn't exist
        std::fs::create_dir_all(&app_support_dir)
            .map_err(|e| format!("Failed to create app data directory:
{}", e))?.;

        let db_path = app_support_dir.join("database.db");
        Ok(db_path.to_string_lossy().to_string())
    }
}

```

4. Rust ↔ Spring Boot (Process Management & HTTP)

Starting Spring Boot from Rust

```
// src-tauri/src/lib.rs
#[cfg(not(debug_assertions))]
fn start_spring_boot_backend(app_handle: tauri::AppHandle, port: u16) ->
Result<(), String> {
    println!("Starting Spring Boot backend on port {}....", port);

    // Get database path
    let db_path = get_database_path(&app_handle)?;

    // Get bundled JAR path
    let resource_path = app_handle
        .path()
        .resource_dir()
        .map_err(|e| format!("Failed to get resource directory: {}", e))?;
    let backend_dir = resource_path.join("resources").join("backend-
spring");
    let jar_path = backend_dir.join("app.jar");

    if !jar_path.exists() {
        return Err(format!("JAR file not found at {:?}", jar_path));
    }

    // Launch Spring Boot with dynamic port and database path
    let child = Command::new("java")
        .current_dir(&backend_dir)
        .arg("-jar")
        .arg(&jar_path)
        .arg(format!("--server.port={}", port))
        .arg(format!("--spring.datasource.url=jdbc:sqlite:{}", db_path))
        .spawn()
        .map_err(|e| format!("Failed to start Spring Boot backend: {}",
e))?;

    // Store process handle for cleanup
    if let Some(process_mutex) = SPRING_BOOT_PROCESS.get() {
        *process_mutex.lock().unwrap() = Some(child);
    }

    Ok(())
}
```

Health Check from Rust

```
// src-tauri/src/lib.rs
#[tauri::command]
async fn check_backend_health() -> Result<bool, String> {
    let port = BACKEND_PORT
        .get()
```

```

        .ok_or_else(|| "Backend port not initialized".to_string())?;

    let client = request::Client::new();
    match client
        .get(format!("http://localhost:{}", port))
        .send()
        .await
    {
        Ok(response) => Ok(response.status().is_success()),
        Err(_) => Ok(false),
    }
}

```

Dynamic Port Selection (Production)

```

// src-tauri/src/lib.rs
#[cfg(not(debug_assertions))]
fn find_available_port() -> Result<u16, String> {
    use rand::Rng;
    use std::net::TcpListener;

    let mut rng = rand::thread_rng();

    // Try up to 100 random ports in range 10000-60000
    for _ in 0..100 {
        let port = rng.gen_range(10000..=60000);
        if TcpListener::bind(("127.0.0.1", port)).is_ok() {
            return Ok(port);
        }
    }

    Err("Could not find an available port".to_string())
}

```

5. Frontend ↔ Rust (Tauri IPC)

Rust Side - Command Definitions

```

// src-tauri/src/lib.rs

// Command to execute a script in a visible terminal
#[tauri::command]
async fn run_script(command: String) {
    println!("Running script: {}", command);
    open_terminal_with_command(command);
}

// Command to get the backend port

```

```
#[tauri::command]
fn get_backend_port() -> Result<u16, String> {
    BACKEND_PORT
        .get()
        .copied()
        .ok_or_else(|| "Backend port not initialized".to_string())
}

// Command to execute shell commands silently
#[tauri::command]
async fn execute_command(command: String) {
    run_terminal_command(command);
}

// Register commands in Tauri
pub fn run() {
    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![
            run_script,
            execute_command,
            get_backend_port,
            check_backend_health,
        ])
        .setup(|app| {
            // ... initialization logic
            Ok(())
        })
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Frontend Side - Invoking Rust Commands

```
// src/App.tsx
import { invoke } from "@tauri-apps/api/core";

function App() {
    const dispatch = useAppDispatch();

    // Fetch backend port on mount (production only)
    useEffect(() => {
        if (!import.meta.env.DEV) {
            const fetchBackendPort = async () => {
                try {
                    const port = await invoke<number>("get_backend_port");
                    dispatch(configSlice.actions.setBackendPort(port));
                    console.log("Backend running on port:", port);
                } catch (error) {
                    console.error("Failed to get backend port:", error);
                }
            };
            fetchBackendPort();
        }
    }, []);
}
```

```

    }
  }, [dispatch]);
}

```

```

// src/app-component/ScriptsColumn/ScriptItem.tsx
import { invoke } from "@tauri-apps/api/core";

const handleRun = async () => {
  try {
    // Opens terminal and executes script
    await invoke("run_script", { command: script.command });
  } catch (error) {
    console.error("Failed to run script:", error);
  }
};

```

```

// src/app-component/FolderColumn/FolderColumn.tsx
const handleOpenBackendApi = async () => {
  const url = `http://localhost:${backendPort}/api`;
  const isMac = navigator.userAgent.includes("Mac");
  const command = isMac ? `open "${url}"` : `start "${url}"`;

  await invoke("run_script", { command });
};

```

6. Frontend ↔ Native Events (Tauri Event System)

Rust Side - Emitting Events

```

// src-tauri/src/lib.rs
#[cfg(target_os = "macos")]
fn setup_menu_handlers(app: &tauri::App) {
  use tauri::Emitter;

  app.on_menu_event(move |app, event| {
    if event.id() == "toggle_dark_mode" {
      if let Some(window) = app.get_webview_window("main") {
        // Emit event to frontend
        window.emit("toggle-dark-mode", ()).unwrap_or_else(|e| {
          eprintln!("Failed to emit toggle-dark-mode event: {}",
e);
        });
      }
    }
  });
}

```

Frontend Side - Listening to Events

```
// src/App.tsx
import { listen } from "@tauri-apps/api/event";

useEffect(() => {
  const unlisten = listen("toggle-dark-mode", async () => {
    const newDarkMode = !darkMode;
    if (appStateData) {
      updateAppState({ ...appStateData, darkMode: newDarkMode });
    }

    // Apply immediately for better UX
    if (newDarkMode) {
      document.documentElement.classList.add("dark");
      await invoke("set_title_bar_color", { isDark: true });
    } else {
      document.documentElement.classList.remove("dark");
      await invoke("set_title_bar_color", { isDark: false });
    }
  });

  return () => {
    unlisten.then((fn) => fn());
  };
}, [darkMode, updateAppState, appStateData]);
```

Data Flow Examples

Example 1: Creating a New Folder

```
User clicks "New Folder"
  ↓
Frontend (FolderColumn.tsx)
  → createFolder({ name: "My Folder" })
  ↓
RTK Query (folderApi.ts)
  → POST http://localhost:{port}/folders
  ↓
Spring Boot (FolderController.kt)
  → folderRepository.save(newFolder)
  ↓
Database (SQLite via JPA)
  → INSERT INTO scripts_folder ...
  ↓
Response flows back up the chain
  ↓
```

```
RTK Query invalidates 'Folder' tag
↓
UI automatically refetches and updates
```

Example 2: Executing a Script

```
User clicks "Execute" on a script
↓
Frontend (ScriptItem.tsx)
  → invoke('run_script', { command: "echo Hello" })
  ↓
Rust (lib.rs)
  → open_terminal_with_command(command)
  ↓
macOS Terminal.app opens
  → Executes the shell command
  → User sees output in terminal
```

Example 3: Application Startup (Production)

```
User launches .app bundle
↓
Rust (lib.rs::run())
  1. init_runtime() - Setup async runtime
  2. init_db() - Prisma creates/migrates database
  3. init_spring_boot() - Find random port, start Java backend
  4. setup_macos_window() - Configure native window
  5. setup_macos_menu() - Create menu bar
  ↓
Frontend (BackendLoadingScreen.tsx)
  → Polls check_backend_health() every 1s
  ↓
Backend becomes ready
  ↓
Frontend (App.tsx)
  → Fetches backend port
  → Sets backendPort in Redux
  ↓
All queries start (skip: !backendPort becomes false)
  ↓
UI loads with data from database
```

Key Features

1. Folder & Script Organization

- Create, rename, delete folders
- Create, edit, delete scripts
- Drag-and-drop reordering for both folders and scripts
- Persistent order maintained in database

2. Script Execution

- Double-click or click "Execute" to run scripts
- Opens in native Terminal.app (macOS)
- Preserves shell environment (\$PATH, aliases, etc.)

3. Dark Mode Support

- System-integrated dark mode
- Toggle via menu (Cmd+Shift+D)
- Persists across sessions
- Native title bar color updates

4. Native macOS Integration

- Transparent title bar with traffic lights
- Custom menu bar with keyboard shortcuts
- Cmd+Q to quit, Cmd+Y for redo
- Standard text editing shortcuts

5. Development vs Production

- **Dev:** Fixed port (7070), manual backend start
- **Production:** Random port, automatic backend bundling
- Environment-specific database paths
- No backend startup delay in dev mode

Running the Application

Development Mode

```
# Terminal 1: Start Spring Boot backend manually
cd backend-spring
./gradlew bootRun

# Terminal 2: Start Tauri dev server
yarn tauri dev
```

Production Build

```
# Build Spring Boot JAR
cd backend-spring
```

```
./gradlew fatJar

# Copy JAR to resources
mkdir -p ../src-tauri/resources/backend-spring
cp build/libs/*-fat.jar ../src-tauri/resources/backend-spring/app.jar

# Build Tauri app
cd ..
yarn tauri build
```

Key Dependencies

Frontend

- `@tauri-apps/api` - Tauri JavaScript bindings
- `@reduxjs/toolkit` - State management
- `@dnd-kit/core` - Drag and drop
- `tailwindcss` - Styling
- `react-router-dom` - Routing (if needed)

Backend

- `spring-boot-starter-web` - REST API
- `spring-boot-starter-data-jpa` - Database ORM
- `sqlite-jdbc` - SQLite driver
- `hibernate-community-dialects` - SQLite dialect

Rust

- `tauri` v2 - Desktop framework
- `serde` - JSON serialization
- `tokio` - Async runtime
- `reqwest` - HTTP client
- `prisma-client-rust` - Database client
- `rand` - Random port generation

Database Schema

```
AppState (application_state)
├── id (PRIMARY KEY)
├── last_opened_folder_id (NULLABLE)
├── dark_mode (BOOLEAN)
├── created_at
└── created_at_hk

ScriptsFolder (scripts_folder)
├── id (PRIMARY KEY)
```



```
├── name
├── order_index
├── created_at
├── created_at_hk

ShellScript (shell_script)
├── id (PRIMARY KEY)
├── name
├── command
├── order_index
├── folder_id (FOREIGN KEY → ScriptsFolder)
├── created_at
├── created_at_hk
```

Relationships:

- **ScriptsFolder** has many **ShellScript** (one-to-many)
- **ShellScript** belongs to one **ScriptsFolder** (with CASCADE delete)

Design Decisions

1. Why Spring Boot + Rust?

- Spring Boot: Rich ecosystem for REST APIs and database management
- Rust: Native performance, safe concurrency, Tauri integration

2. Why SQLite?

- Embedded database (no separate server)
- Single-file portability
- Perfect for desktop applications

3. Why Prisma in Rust?

- Type-safe database access
- Automatic schema migrations
- Single source of truth for schema

4. Why RTK Query?

- Automatic caching and invalidation
- Optimistic updates for better UX
- Reduces boilerplate vs manual fetch

5. Why Dynamic Port in Production?

- Avoids port conflicts with other apps
- Allows multiple instances if needed
- More secure (unpredictable port)



Notes

- **Flyway removed:** Prisma manages all schema changes
 - **Hibernate DDL disabled:** `ddl-auto: none` to prevent conflicts
 - **Database path:** Dev uses project dir, Production uses Application Support
 - **Backend lifecycle:** Managed by Rust, auto-starts in production
 - **Query skipping:** Queries wait for backend port before executing
 - **Error handling:** Rust uses `Result<>` to prevent panics in Objective-C context
-



Future Enhancements

- ☐ Script templates library
 - ☐ Script history/audit log
 - ☐ Environment variable management per script
 - ☐ Script output capture and display
 - ☐ Search/filter functionality
 - ☐ Import/export script collections
 - ☐ Windows/Linux support
 - ☐ Cloud sync (optional)
-

Generated: November 2, 2025