Repository

- https://github.com/machingclee/2025-10-27-shell-script-manager-tauri/tree/main/src

Overview

The application uses Tauri's `WebviewWindow` API to create separate native windows. Each window runs its own webview with shared Redux state, allowing for synchronized settings like dark mode.

Step-by-Step Guide

Steps

Create Your Custom Component

Create your component in the appropriate directory (e.g., `src/app-component/`):

```
// src/app-component/MyCustomComponent.tsx
export default function MyCustomComponent({ someId }: { someId?: number })
{
    return (
        <div className="h-screen w-screen bg-white dark:bg-gray-900">
            <h1>My Custom Window</h1>
            {/* Your component content */}
        </div>
    );
}
```

Important: Use Tailwind's `dark:` prefix for dark mode styling.

Create Window Entry Point

Create an entry file in `src/subwindows/`:

```
// src/subwindows/my-window-entry.tsx
import React, { useEffect, useState } from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import { store } from "../store/store";
import MyCustomComponent from "../app-component/MyCustomComponent";
import { BackendLoadingScreen } from "../components/BackendLoadingScreen";
import { appStateApi } from "../store/api/appStateApi";
import { useAppSelector } from "../store/hooks";
import "../index.css";

function MyWindowContent() {
    const [someId, setSomeId] = useState<number | undefined>(undefined);
    const backendPort = useAppSelector((s) => s.config.backendPort);
```

```
    // Fetch app state to get dark mode setting
    const { data: appState } =
appStateApi.endpoints.getAppState.useQuery(undefined, {
        skip: !backendPort,
    });

    const darkMode = appState?.darkMode ?? false;

    useEffect(() => {
        // Get parameters from URL query string
        const params = new URLSearchParams(window.location.search);
        const id = params.get("someId");
        if (id) {
            setSomeId(parseInt(id, 10));
        }
    }, []);

    // Apply dark mode class to html element
    useEffect(() => {
        if (darkMode) {
            document.documentElement.classList.add("dark");
        } else {
            document.documentElement.classList.remove("dark");
        }
    }, [darkMode]);

    return <MyCustomComponent someId={someId} />;
}

ReactDOM.createRoot(document.getElementById("root") as
HTMLElement).render(
    <React.StrictMode>
        <Provider store={store}>
            <BackendLoadingScreen>
                <MyWindowContent />
            </BackendLoadingScreen>
        </Provider>
    </React.StrictMode>
);
```

Key Points:

- Wrap with `Provider` and pass the same `store` instance to share Redux state
- Use `BackendLoadingScreen` to ensure backend is ready
- Fetch `appState` using RTK Query to get dark mode setting
- Apply `dark` class to `document.documentElement` based on dark mode
- Extract URL parameters to pass data to your component

Create HTML Entry File

Create an HTML file in the project root:

```
<!-- my-window.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <link rel="icon" type="image/svg+xml" href="/vite.svg" />
        <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
        <title>My Custom Window</title>
    </head>
    <body>
        <div id="root"></div>
        <script type="module" src="/src/subwindows/my-window-entry.tsx">
</script>
    </body>
</html>
```

Update Vite Configuration

Add your new window to the Vite build configuration:

```
// vite.config.ts
export default defineConfig({
    // ... other config
    build: {
        rollupOptions: {
            input: {
                main: "index.html",
                markdown: "markdown-window.html",
                mywindow: "my-window.html", // Add your window here
            },
        },
    },
});
```

Update Tauri Permissions

Add window creation permissions to `src-tauri/tauri.conf.json`:

```
{
    "app": {
        "security": {
            "capabilities": [
                {
                    "identifier": "main-window",
                    "windows": ["main"],
                    "permissions": [
```

```
                        "core:webview:allow-create-webview-window",
                        "core:window:allow-get-all-windows",
                        "core:window:allow-set-focus"
                    ]
                },
                {
                    "identifier": "my-windows",
                    "windows": ["my-*"],
                    "permissions": ["core:window:default",
"core:webview:default"]
                }
            ]
        }
    }
}
```

Required permissions for main window:

- `core:webview:allow-create-webview-window` - Create new child windows
- `core:window:allow-get-all-windows` - Get list of all windows (for auto-focus feature)
- `core:window:allow-set-focus` - Bring subwindows to front when main window gains focus

Open the Window from Your Code

Use Tauri's `WebviewWindow` API to open your new window:

```
import { WebviewWindow } from "@tauri-apps/api/webviewWindow";

const handleOpenMyWindow = async (id: number) => {
    try {
        const windowLabel = `my-${id}`;
        const url = `/my-window.html?someId=${id}`;

        const newWindow = new WebviewWindow(windowLabel, {
            url,
            title: "My Custom Window",
            width: 1200,
            height: 800,
            // parent: "main", // Optional - see "Window Parent
Relationship" section below
        });

        // Optional: Listen for events
        await newWindow.once("tauri://error", (e) => {
            console.error("Error creating window:", e);
        });
    } catch (error) {
        console.error("Failed to open window:", error);
    }
};
```

Passing State via Query Parameters

You can pass additional state to child windows via URL query parameters to initialize the component's state.

Example: Edit Mode Parameter

```
// src/lib/subwindowPaths.ts
export const getSubwindowPaths = {
    markdown: (scriptId: number, editMode: boolean = false): string => {
        const path = "src/subwindows/markdown-window.html";
        const queryParams = `scriptId=${scriptId}${editMode ?
"&editMode=true" : ""}`;
        const url = import.meta.env.DEV
            ? `http://localhost:1420/${path}?${queryParams}`
            : `/${path}?${queryParams}`;
        return url;
    },
};
```

Opening windows with different initial states:

```
// Open in view mode (default)
const handleViewClick = () => {
    const url = getSubwindowPaths.markdown(scriptId, false);
    new WebviewWindow(`markdown-${scriptId}`, {
        url,
        title: `View: ${scriptName}`,
        // ... other options
    });
};

// Open in edit mode
const handleEditClick = () => {
    const url = getSubwindowPaths.markdown(scriptId, true);
    new WebviewWindow(`markdown-${scriptId}`, {
        url,
        title: `Edit: ${scriptName}`,
        // ... other options
    });
};
```

Reading query parameters in the child window component:

```
export default function MyComponent({ someId }: { someId: number |
undefined }) {
    // Read query parameter from URL
    const urlParams = new URLSearchParams(window.location.search);
    const editModeFromUrl = urlParams.get("editMode") === "true";
```

```
    // Initialize state with the query parameter value
    const [isEditMode, setIsEditMode] = useState(editModeFromUrl);

    // Optional: Initialize other state when starting in special mode
    useEffect(() => {
        if (editModeFromUrl && data) {
            // Perform additional initialization for edit mode
            setSomeState(data.someValue);
        }
    }, [data, editModeFromUrl]);

    return <div>{isEditMode ? <EditView /> : <ReadOnlyView />}</div>;
}
```

Benefits:

- Different entry points can open the same window with different initial configurations
- Avoids the need to pass state through events after window creation
- Cleaner URL-based state initialization
- Supports browser-like navigation patterns

Window Parent Relationship

The `parent: "main"` property is **optional** and has trade-offs:

With `parent: "main"`:

- ✅ Child window automatically stays above parent window (z-order)
- ✅ OS window managers may group windows together
- ❌ Child window moves when you drag the parent window (coupled movement)
- ❌ Minimizing parent may also minimize children (OS-dependent)

Without `parent: "main"` (independent windows):

- ✅ Windows move independently - drag main window without moving child
- ✅ More flexible window management
- ❌ Child windows don't automatically stay above parent
- ❌ May feel less "attached" to the main app

Important: The `parent` property is **NOT required** to prevent child windows from closing the main app. This is handled by the Rust code checking window labels (see "App Shutdown vs Child Window Close" section).

Dark Mode Synchronization

Dark mode synchronization happens automatically through:

1. **Shared Redux Store**: Both windows use the same store instance
2. **RTK Query**: Fetches `appState` which includes `darkMode` setting
3. **Class Application**: Applies/removes `dark` class on `document.documentElement`

```
// This pattern syncs dark mode
const { data: appState } =
appStateApi.endpoints.getAppState.useQuery(undefined, {
    skip: !backendPort,
});

const darkMode = appState?.darkMode ?? false;

useEffect(() => {
    if (darkMode) {
        document.documentElement.classList.add("dark");
    } else {
        document.documentElement.classList.remove("dark");
    }
}, [darkMode]);
```

When dark mode changes in the main window, RTK Query automatically updates all subscribed components across all windows, triggering the class update.

Inter-Window Communication

Important: Each Tauri window runs in its own JavaScript context with its own Redux store instance. While both windows import the same store file, they each create separate instances in their own memory space - they do NOT share the same store.

When a child window makes changes that should be reflected in the main window (or other windows), you must use Tauri's event system to communicate between windows.

Why Event-Based Communication is Needed

```
Main Window Memory Space          Child Window Memory Space
├── Redux Store (Instance A)      ├── Redux Store (Instance B)
├── RTK Query Cache               ├── RTK Query Cache
└── React Components              └── React Components

❌ Changes in Instance B do NOT automatically update Instance A
✅ Use emit/listen to notify other windows of changes
```

Implementation Pattern

In the child window (when making changes):

```
import { emit } from "@tauri-apps/api/event";
import { scriptApi } from "@/store/api/scriptApi";

const [updateData] = scriptApi.endpoints.updateData.useMutation();
```

```
const handleSave = async () => {
    // Save changes
    await updateData(updatedData).unwrap();

    // Invalidate cache in this window
    dispatch(scriptApi.util.invalidateTags([{ type: "Script", id: dataId
}]));

    // Notify other windows to refresh their cache
    await emit("data-updated", { id: dataId });
};
```

In the main window (listening for changes):

```
import { listen } from "@tauri-apps/api/event";
import { scriptApi } from "@/store/api/scriptApi";

useEffect(() => {
    const unlisten = listen<{ id: number }>("data-updated", ({ payload })
=> {
        console.log("Data updated in child window, refreshing cache");
        dispatch(scriptApi.util.invalidateTags([{ type: "Script", id:
payload.id }]));
    });

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

Key Points:

- **Emit events** from the window making changes to notify others
- **Listen for events** in windows that need to stay synchronized
- **Invalidate RTK Query cache** in both the sending and receiving windows
- Use specific event names like `"markdown-updated"`, `"folder-created"`, etc.
- Include necessary data in the event payload (IDs, etc.)

Example: Markdown Editor

The markdown editor demonstrates this pattern:

- Child window edits markdown name/content
- After saving, emits `"markdown-updated"` event with script ID
- Main window listens for event and invalidates its cache
- Main window UI automatically refreshes with updated data

This pattern ensures all windows stay in sync despite having separate Redux store instances.

Example: Markdown Window

See the existing implementation:

- Entry: src/subwindows/markdown-window-entry.tsx
- HTML: markdown-window.html
- Component: src/app-component/ScriptsColumn/MarkdownEditor.tsx
- Opener: src/app-component/ScriptsColumn/MarkdownItem.tsx

Window Lifecycle & Closing Behavior

Closing Child Windows

Child windows can be closed normally using `getCurrentWindow().close()`:

```
import { getCurrentWindow } from "@tauri-apps/api/window";

const handleCloseWindow = () => {
    getCurrentWindow().close();
};
```

Rust Adjustment for App Shutdown vs Child Window Close

The application distinguishes between closing the main window and closing child windows:

- **Main Window Close**: Triggers backend shutdown, shows "Shutting down..." overlay, and cleans up resources
- **Child Window Close**: Closes immediately without affecting the main application

This is handled in `src-tauri/src/lib.rs`:

```
.on_window_event(|window, event| {
    if let tauri::WindowEvent::CloseRequested { api, .. } = event {
        // Only handle close for the main window
        if window.label() != "main" {
            // Allow child windows to close normally
            return;
        }

        // Main window cleanup logic...
    }
})
```

Key Points:

- The `on_window_event` handler filters events by window label
- Only the main window close (label = "main") triggers the `"app-closing"` event
- Child windows close immediately without cleanup delays
- **The `parent: "main"` property is NOT required** for this safety mechanism

Closing Overlay

The `AppClosingOverlay` component shows a "Shutting down..." message only when the main window is closing:

```
// src/components/AppClosingOverlay.tsx
// Listens to "app-closing" event, which is only emitted for main window
```

Because the Rust handler filters by window label (`window.label() != "main"`), closing child windows will NOT trigger this overlay, regardless of whether they have a `parent` property set.

Window Z-Order Management

The application uses natural OS window management for z-ordering:

Default Behavior:

- Click any window → That window comes to the front
- Windows stack based on which one you clicked most recently
- Each window (main or child) can be independently brought to the top
- No automatic stacking or forced ordering between windows

Why This Approach:

- ✅ Natural, predictable window behavior
- ✅ Users can organize windows as they prefer
- ✅ No flickering or forced focus changes
- ✅ Works seamlessly with OS window management (Mission Control, Spaces, etc.)

Alternative: Auto-Focus (Not Recommended):

You could implement auto-focus to bring all subwindows forward when clicking the main window:

```
// ⚠ Not recommended - can cause flickering and unexpected behavior
useEffect(() => {
    const mainWindow = getCurrentWindow();

    const unlisten = mainWindow.onFocusChanged(async ({ payload: focused
}) => {
        if (focused) {
            const allWindows = await getAllWebviewWindows();
            const subwindows = allWindows.filter((w) => w.label !==
"main");
            for (const subwindow of subwindows) {
                await subwindow.setFocus();
            }
        }
    });

    return () => {
```

```
            unlisten.then((fn) => fn());
        };
    }, []);
```

Issues with Auto-Focus:

- ❌ Causes flickering as windows are programmatically focused
- ❌ Prevents natural window ordering based on user clicks
- ❌ Main window may end up behind subwindows
- ❌ Interferes with multi-window workflows

Best Practice: Let the OS handle window z-ordering naturally. Users can click whichever window they want to bring to front.

Troubleshooting

Window Won't Open

- Check Tauri permissions in `tauri.conf.json`
- Verify the window label pattern matches the capability pattern
- Check browser console for permission errors

Closing Child Window Triggers App Shutdown

- Check that `src-tauri/src/lib.rs` filters events by `window.label() != "main"`
- Ensure the child window label is NOT "main" (use patterns like `my-*`, `markdown-*`, etc.)
- Verify window label follows the pattern defined in capabilities

Child Window Moves When Dragging Main Window

- This happens when `parent: "main"` is set in `WebviewWindow` options
- Remove the `parent` property for independent window movement
- Note: Removing `parent` is safe - the Rust close handler checks window labels, not parent relationships

Dark Mode Not Syncing

- Ensure Redux Provider wraps your component
- Verify `appStateApi` query is active (not skipped)
- Check that `dark` class is applied to `document.documentElement`, not body

Build Fails

- Ensure HTML file is added to `vite.config.ts` rollupOptions.input
- Verify all import paths are correct relative to the entry file location

Content Security Policy (CSP)

Tauri applications run in a secure webview environment that enforces a Content Security Policy (CSP) to prevent security vulnerabilities. When creating new windows or adding features that interact with external resources, you may need to adjust the CSP.

When to Configure CSP

You need to modify the CSP when:

1. **Backend Communication**: Your frontend needs to fetch data from a local backend server (e.g., Spring Boot on `localhost`)
2. **External API Calls**: Your app makes requests to external APIs (e.g., OpenAI, weather APIs)
3. **Dynamic Styles**: Using CSS-in-JS libraries that inject styles dynamically (e.g., MUI with Emotion, styled-components)
4. **WebSockets**: Establishing WebSocket connections to local or remote servers
5. **Images from External Sources**: Loading images from HTTPS URLs or data URIs
6. **Web Workers**: If your dependencies use web workers (though this is rare in typical React apps)

CSP Configuration Location

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

```
{
    "app": {
        "security": {
            "csp": "default-src 'self' tauri: http://tauri.localhost; ..."
        }
    }
}
```

Common CSP Directives

Backend Communication (Critical):

```
"csp": "... connect-src 'self' tauri: http://tauri.localhost
http://localhost:* ws://localhost:*; ..."
```

- `http://localhost:*` - Allows `fetch()` requests to your local backend on any port
- `ws://localhost:*` - Allows WebSocket connections to local backend
- **Why needed**: Tauri runs on `http://tauri.localhost`, but your backend runs on `http://localhost:port`. These are different origins, so CORS applies.

External API Calls:

```
"csp": "... connect-src 'self' tauri: http://tauri.localhost
https://api.example.com; ..."
```

- Add specific HTTPS domains for each external API you call
- Example: `https://api.openai.com` for OpenAI API calls

Dynamic Styles (MUI/Emotion):

```
"csp": "... style-src 'self' tauri: http://tauri.localhost 'unsafe-
inline'; ..."
```

- `'unsafe-inline'` - Required for libraries that inject `<style>` tags dynamically
- **Needed for**: MUI (Emotion), styled-components, CSS-in-JS libraries

External Images:

```
"csp": "... img-src 'self' tauri: http://tauri.localhost data: https:;
..."
```

- `data:` - Allows inline base64-encoded images
- `https:` - Allows images from any HTTPS source (more permissive, use specific domains for stricter security)

Complete Example

Here's a typical CSP for an app with a local backend and MUI:

```
"security": {
    "csp": "default-src 'self' tauri: http://tauri.localhost; connect-src
'self' tauri: http://tauri.localhost http://localhost:* ws://localhost:*;
font-src 'self' tauri: http://tauri.localhost; img-src 'self' tauri:
http://tauri.localhost data: https:; style-src 'self' tauri:
http://tauri.localhost 'unsafe-inline'; script-src 'self' tauri:
http://tauri.localhost 'unsafe-inline' 'unsafe-eval'"
}
```

Breakdown:

- `default-src 'self' tauri: http://tauri.localhost` - Base policy for all resources
- `connect-src ... http://localhost:*` - Backend API calls + WebSockets
- `font-src 'self' tauri: http://tauri.localhost` - Local fonts only
- `img-src ... data: https:` - Local images + base64 + HTTPS images
- `style-src ... 'unsafe-inline'` - Dynamic styles for MUI/Emotion
- `script-src ... 'unsafe-eval'` - Some frameworks need this (use cautiously)

Debugging CSP Issues

Symptoms:

- `fetch()` calls fail with CORS errors
- Styles don't apply or look broken
- Console shows CSP violation errors: `"Refused to load ... because it violates the following Content Security Policy directive..."`

Debugging Steps:

1. **Check Browser Console**: Look for CSP violation messages
2. **Identify the Blocked Resource**: Note which directive is violated (e.g., `connect-src`, `style-src`)
3. **Add to CSP**: Update the relevant directive in `tauri.conf.json`
4. **Rebuild**: CSP changes require a full rebuild (`yarn bundle`)
5. **Test**: Verify the resource now loads correctly

Example Error:

```
Refused to connect to 'http://localhost:8080/api/data' because it violates
the following Content Security Policy directive: "connect-src 'self'
tauri: http://tauri.localhost".
```

Solution: Add `http://localhost:*` to `connect-src`.

Security Best Practices

1. **Be Specific**: Instead of `https:`, list specific domains: `https://api.example.com https://cdn.example.com`
2. **Avoid `unsafe-eval`**: Only use if absolutely necessary (some frameworks require it)
3. **Minimize `unsafe-inline`**: Only add for `style-src` if using CSS-in-JS libraries
4. **Test Thoroughly**: Test all features after CSP changes to ensure nothing is blocked
5. **Development vs Production**: CSP applies to both - test in production builds

Common Pitfall: Backend Port Changes

If your backend port changes dynamically, `http://localhost:*` (with wildcard) is the safest approach. Alternatively, use a fixed port in your backend configuration.