Project Overview

Name: Tauri application example
Description: Tauri application that opens a pre-specified URL in the default window and persists its own state between sessions in the user data folder.

Project requirements

- The builded client should be able to read the URL from as command line argument and to open that URL in the main app window context.
- The application should persist its own state (cookies, sessions, cache, window size and location, etc.) between sessions in a file that's located in the default user data folder.

Objectives

Example Tauri application should be able to read the URL from the command line and store its own state in a file located in the user's home directory.

Deliverables

- Example application for Windows that can open a pre-specified URL when run, and persist own state in a file
- Documentation on how to build and run the example client
- The source code for the project

Functional requirements

Builded client should be able to open a pre-specified URL from the command line.

Non-functional Requirements

Builded client should be able to store its own state (cookies, sessions, cache, window size and location, etc.) in a file located in the user's home directory.

Design

The architectural design of this example uses the technology embedded web content in native applications - WebView2. The application, on startup, reads an URL from the user's command line and then displays the contents of that URL in the application's main window. To save and later restore the state, the application stores its data in the default user data folde.

Worklog

Tauri framework represents a proxy layer that helps in communication between web content and the host operation system. It provides an abstraction layer on top of operation system calls and a unified API for interacting with different operating systems using a single codebase.

The plan

1. Get familiarize with the Tauri framework (tools, build and debug process, WebView2) + environmental setup.
2. Create a lightweight client that meets the requirements on the host OS (Windows 11) directly.
3. Test builded client under different host operating systems.

[tauti-app]

The initial assumption is to split development process into 2 parts:

- Build an example Tauri application that met the requirements on the host OS directly.
- Build receipt in form of Dockerfile that is able to build such Tauri application executables for Windows (x86_64)

Initial assumption was to build a Tauri app client which fontend is based on React. During the Tauri application build process, the React application client bundles it with a WebView2 instance created on startup using Tauri “commands”.

Thus, we can separate the process of launching a web application and the rest of the functionality:

- saving and restoring the state, for example through localStorage (appWindow.onMoved, appWindow.onResized, etc). Initial assumption: localStorage is transparently mapped by Tauri to WebView2 UDF.
- Navigating to url using WebView2 instance “window.location.url” property

Thus, We can assemble a nesting "matryoshka" doll where the core is the Rust backend and the shell is frontend web application based on React and rendered by default WebView2 and the glue between them is in form of "@tauri/api" package

However, this method has its advantages and disadvantages:

1. It is possible to explicitly save data in files using the @tauri-apps/api library API ("path" module). Thus, it is possible to save the state of the application between working sessions.
2. However, there is a problem in the form of collecting all possible state paratters (with the size and position of the window is simple, but with cookies, cache, and other structures that WebView2 uses - it will be more difficult)

[tauri-app-vanilla]

Alternative solution was to implement all functional requirements through Tauri (backend) using Rust. So we could just set a WebView2 instance to render the remote frontend instead of the local one.

I've found a plugin for Tauri window state management "**tauri-plugin-window-state**", but at this moment i'm not sure if it's saving all the state using a file in the host OS used home directory, need to check that.

And there are no disadvantages in this method.:

1. There is a "tauri-plugin-window-state" plugin that holds all the necessary information and does not need to be collected: save (on app close) and restore (on app start) - the plugin does everything on its own.
2. An instance of the WebView2 class of the Tauri UDF application. WebView2 applications use User Data Folders (UDFs) to store browser's data such as sessions, cookies, permissions, and cached resources.

The "tauri-plugin-window-state" plugin uses the Tauri api folder and the path.appConfigDir() module function to store the application's window state structure:

```
struct WindowState {
    width: f64,
    height: f64,
    x: i32,
    y: i32,
    maximized: bool,
    visible: bool,
    decorated: bool,
    fullscreen: bool,
}
```

"path.appConfigDir()" resolves to ${configDir}/${bundleIdentifier} where bundleIdentifier is the value that "**tauri.bundle.identifier**" property is configured in **tauri.conf.json** and ${configDir} resolves to $XDG_CONFIG_HOME or $HOME/.config, which is user data folder (UDF)

The default path to the UDF folder of the WebView2 component can be changed using the WEBVIEW2_USER_DATA_FOLDER environment variable which should be set in the Tauri backend before the Tauri app (and default WebView2 instance) initialization. But the WebView2 instance by default stores its own state in the Windows default user data folder which is $HOME.

Due to the heuristic approach used in the current workflow I've tried both approaches and due to example application limited requirements set (and additional workload if using [tauri-app] approach) I decided to stick with the second vanilla approach [tauri-app-vanilla].

Command line arguments

We have several ways to order a WebView2 to render external content from the URL. The easiest way is the Tauri application window config (WindowConfig) located in **src-tauri/tauri.conf.json** with "**tauri > windows[0] > url**" property. However, this cannot be done dynamically (from the command line of the compiled application) - only before the application build phase.

Also, we have the following requirement:

"*Write a Tauri client that, when run, opens a pre-specified URL. It should read the URL from command line*"

To satisfy this requirement, we need to read the command line argument when the application starts and inject the url into the WebView2 during the startup process. The specified url might be accessible by “url” cli argument name. It's even better to make the "url" command line argument optional, and if there is no value, no changes will be made to the webview. So, the default url from **tauri.conf.json** will be used, which can be changed manually or during the build process of the application in the Docker container.

Quite a lot of time was spent on solving this problem, since I have no experience with Tauri and the Rust ecosystem in general. But now I don't know more. At this point, I learned how Rust Cargo works, how to initialize a Tauri application and a web content renderer, and I enjoyed implementing the functional paradigm in Rust and using C-style pointers.

Improvements

Due to that fact that we don’t use local Tauri JS frontend it’s would be better to get rid of it by removing all HTML/JS files