What is Tauri?

Tauri is an **open-source framework** for building **small, fast, and secure desktop applications** using web technologies (HTML, CSS, JavaScript/TypeScript). Unlike traditional solutions like Electron, Tauri uses your operating system's native webview (e.g., WebKit on macOS, WebView2 on Windows) instead of bundling a full browser engine.

Key Features & Benefits

1. **Lightweight & Efficient**:
   - Apps are **10–100x smaller** than Electron (e.g., a simple app can be as small as **2MB** vs. Electron’s 100MB+).
   - Minimal memory usage and faster startup times.
2. **Cross-Platform**:
   - Build for **Windows, macOS, and Linux** from a single codebase.
   - Native system integrations (tray icons, file system access, notifications, etc.).
3. **Security-First**:
   - Built with **Rust** (memory-safe language) for the backend.
   - Secure IPC (inter-process communication) between frontend and backend.
   - Reduced attack surface compared to Chromium-based frameworks.
4. **Modern Web Tech**:
   - Use any frontend framework (React, Vue, Svelte, etc.) or vanilla HTML/CSS/JS.
   - Access **system APIs** via JavaScript (e.g., file system, clipboard, hardware).
5. **Native Performance**:
   - Rust backend handles heavy computations, while the webview focuses on UI.
   - Direct access to OS features without performance bottlenecks.
6. **Extensible**:
   - Plugin system for adding native functionality (e.g., SQLite, biometric auth).
   - Integrates with tools like Vite, Webpack, or Rollup.

Why Choose Tauri Over Electron?

Aspect	Tauri	Electron
App Size	2–20 MB	100–300 MB
Memory Usage	Minimal (uses system webview)	High (bundles Chromium)
Security	Rust backend + secure IPC	Larger attack surface (Chromium)
Performance	Native-speed Rust integration	JavaScript-only backend
Flexibility	Modern tooling (Vite, Deno, etc.)	Limited to Node.js ecosystem

Use Cases

Tauri is ideal for:

- **Lightweight apps** where bundle size matters (e.g., utilities, tools).
- **Privacy-focused apps** (password managers, note-taking apps).
- **Cross-platform apps** needing native OS integrations.
- Projects prioritizing **performance** and **security**.

Who Uses Tauri?

Companies like Microsoft, Discord, and Logseq leverage Tauri for its efficiency and security. The framework is **open-source** (MIT/Apache 2.0) and backed by a growing community.

Getting Started

```
npm create tauri-app@latest
```

Learn more: Tauri’s Official Website | GitHub

In short, Tauri combines **web flexibility** with **native performance**, making it a top choice for modern desktop app development. 🚀

Edit this page

Prerequisites

Before you start building your Tauri app, ensure your system meets the following requirements. Tauri relies on several tools and dependencies to function properly. Follow the instructions for your operating system.

1. Supported Operating Systems

Tauri supports the following operating systems:

- **Windows** (10/11, 64-bit)
- **Linux** (Debian/Ubuntu, Fedora, Arch, etc.)
- **macOS** (10.13 or later)

2. Chose your guide based on your operating system

- Windows Guide
- Linux Guide
- MacOS Guide
- **More detailed documentation**

Already have all the prerequisites installed? Great job!

- Getting Started

Edit this page

Tauri Setup Guide for Windows

Step 1: Install Required Software

1. Install Deno (JavaScript/TypeScript Runtime)

- **Website**: https://deno.land/
- **Steps**:
  i. Open PowerShell
  ii. Run the following command:

```
irm https://deno.land/install.ps1 | iex
```

2. Install Rust (Tauri Backend)

- **Website**: https://www.rust-lang.org/
- **Steps**:
  i. Go to the Rust website.
  ii. Click the **"Get Started"** button.
  iii. Click the **"Other Installation Methods"** hyperlink
  iv. Click on `rustup-init.exe` to get the installer.
  v. Run the installer and follow the prompts.
    - When prompted, choose **"Proceed with installation"**.
  vi. Restart your computer after installation.

3. Install WebView2 (Windows WebView Runtime)

- **Website**: https://developer.microsoft.com/en-us/microsoft-edge/webview2/
- **Steps**:
  i. Go to the WebView2 website.
  ii. Download the **Evergreen Standalone Installer**.
  iii. Run the installer and follow the prompts.

4. Install Visual Studio Build Tools (Required for Rust)

- **Website**: https://visualstudio.microsoft.com/visual-cpp-build-tools/
- **Steps**:
  i. Go to the Visual Studio Build Tools website.
  ii. Download the installer.
  iii. Run the installer and:
    - Select **"Desktop development with C++"** workload.
    - Click **"Install"**.

Troubleshooting

1. WebView2 Issues

- Ensure WebView2 is installed by visiting https://developer.microsoft.com/en-us/microsoft-edge/webview2/.

2. Rust/Cargo Errors

- Visit https://www.rust-lang.org/ and reinstall Rust if needed.

3. Deno Permissions

- If Deno throws permission errors, use the `-A` flag to allow all permissions:

```
deno run -A npm:create-tauri-app@latest
```

Final Notes

- **Deno Documentation**: https://deno.land/manual
- **Tauri Documentation**: https://tauri.app/v1/guides/
- **Vite Documentation**: https://vitejs.dev/guide/

Edit this page

Tauri Setup Guide for Linux

Step 1: Install Required Software

Linux (Debian/Ubuntu)

1. **Install Deno**:
   - Open a terminal and run:
     ```
     curl -fsSL https://deno.land/x/install/install.sh | sh
     ```
2. **Install Rust**:
   - Open a terminal and run:
     ```
     curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
     ```
   - Restart your terminal after installation.
3. **Install System Dependencies**:
   - Open a terminal and run:
     ```
     sudo apt update
     sudo apt install -y libwebkit2gtk-4.0-dev \
       build-essential \
       curl \
       wget \
       libssl-dev \
       libgtk-3-dev \
       libayatana-appindicator3-dev
     ```

Troubleshooting

- **Missing Dependencies**:
  - Run:
    ```
    sudo apt update
    sudo apt install -y libwebkit2gtk-4.0-dev build-essential curl wget libssl-dev libgtk-3-dev libayatana-appindicator3-dev
    ```
- **Deno Permissions**:
  - Use the `-A` flag to allow all permissions:
    ```
    deno run -A npm:create-tauri-app@latest
    ```

Final Notes

- **Deno Documentation**: https://deno.land/manual
- **Tauri Documentation**: https://tauri.app/v1/guides/
- **Vite Documentation**: https://vitejs.dev/guide/

Edit this page

Tauri Setup Guide for macOS

Step 1: Install Required Software

1.1 Install Xcode Command Line Tools

1. Open the **Terminal** (press `Cmd + Space`, type `Terminal`, and press Enter).
2. Run the following command:

```
xcode-select --install
```

3. Follow the prompts to install the Xcode Command Line Tools.

1.2 Install Homebrew (Package Manager)

1. Open the **Terminal**.
2. Run the following command:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

3. Follow the prompts to complete the installation.

1.3 Install Deno

1. Open the **Terminal**.
2. Run the following command:

```
brew install deno
```

1.4 Install Rust

1. Open the **Terminal**.
2. Run the following command:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

3. Follow the prompts to complete the installation.
4. Restart your terminal after installation.

Troubleshooting

1. Xcode Command Line Tools Issues

- If `xcode-select --install` fails, download Xcode from the Mac App Store.

2. Rust/Cargo Errors

- Update Rust:

```
rustup update
```

3. Deno Permissions

- If Deno throws permission errors, use the `-A` flag to allow all permissions:

```
deno run -A npm:create-tauri-app@latest
```

Final Notes

- **Deno Documentation**: https://deno.land/manual
- **Tauri Documentation**: https://tauri.app/v1/guides/
- **Vite Documentation**: https://vitejs.dev/guide/

Edit this page

Getting Started

Step 0: Verify Your Setup

After installing the prerequisites, verify your setup by running the following commands in your terminal:

- **Deno**:

```
deno --version
```

- **Rust**:

```
rustc --version
cargo --version
```

- **Node.js** (if installed):

```
node --version
npm --version
```

Step 1: Scaffold the Project

1. Open your terminal.
2. Run the following command to create a new Tauri project with Vite and TypeScript:

```
deno run -A npm:create-tauri-app@latest
```

3. **Follow the prompts**:
   - ✔ Project name · your-tauri-project
   - ✔ Identifier · com.your-tauri-project.app
   - ✔ Choose which language to use for your frontend · TypeScript / JavaScript - (pnpm, yarn, npm, deno, bun)
   - ✔ Choose your package manager · deno
   - ✔ Choose your UI template · Vanilla
   - ✔ Choose your UI flavor · TypeScript

Step 2: Navigate to the Project Folder

1. Move into the project folder:

```
cd your-tauri-project
```

Step 3: Install Tauri API for Deno

1. Install the Tauri API for Deno:

```
deno task tauri add dialog
```

Step 4: Configure Vite

1. Open the `vite.config.ts` file in your project and ensure it looks like this:

```
import { defineConfig } from "vite";
import deno from "vite-plugin-deno";

export default defineConfig({
  plugins: [deno()],
});
```

2. If you don't have `vite-plugin-deno` installed, add it:

```
deno add vite-plugin-deno
```

Step 5: Set Up the Frontend

1. Open the `src/main.ts` file and replace its content with this basic TypeScript example:

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

// Create a button that calls a Tauri command
const app = document.querySelector<HTMLDivElement>("#app")!;
app.innerHTML = `
  <h1>Hello, Tauri + Vite!</h1>
  <button id="greet-btn">Say Hello</button>
  <p id="message"></p>
`;

// Add event listener to the button
const greetBtn = document.querySelector<HTMLButtonElement>("#greet-btn")!;
const message = document.querySelector<HTMLParagraphElement>("#message")!;

greetBtn.addEventListener("click", async () => {
  const response = await invoke("greet", { name: "Tauri User" });
  message.textContent = response as string;
});
```

Step 6: Set Up the Tauri Backend

1. Open the `src-tauri/src/main.rs` file and add a simple command:

```rust
#[tauri::command]
fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}

fn main() {
    tauri::Builder::default()
        .invoke_handler(tauri::generate_handler![greet])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

Step 7: Run the Development Server

1. Install dependecies:

```
deno install
```

2. Start the Tauri development medium:

```
deno task tauri dev
```

This will open a desktop window with your app running.

Step 8: Build the App

1. Build the production version:

```
deno task build
```

2. The installer will be located in:

   - **Windows**: `.msi` file in `src-tauri/target/release/bundle/msi/`.
   - **Linux**: `.deb` or `.AppImage` file in `src-tauri/target/release/bundle/`.
   - **macOS**: `.dmg` file in `src-tauri/target/release/bundle/dmg/`.

Project Structure

```
tauri-vite-app/
├─ src/            # Vite frontend (TypeScript)
│  ├─ main.ts      # Entry point
│  └─ assets/      # Static assets
├─ src-tauri/      # Tauri backend (Rust)
│  ├─ src/         # Rust source files
│  └─ tauri.conf.json # Tauri configuration
├─ deno.json       # Deno configuration
└─ vite.config.ts  # Vite configuration
```

Edit this page

Build Your Own Text Editor

"Are you excited?😄"

"Rust is the bridge between fearless system control and modern application development."

This project isn't just about building a text editor—it's a **deep dive into full-stack systems programming**. You'll stitch together:

Core Concepts Unlocked

- **Rust's Memory Safety**: Direct file I/O with `std::fs` (no segfaults, no leaks)
- **Cross-Language Pipelines**: TypeScript ↔ Rust communication via Tauri's IPC
- **System-Level APIs**: Native file dialogs, directory traversal, OS permissions
- **State Management**: Sync UI with filesystem changes in real time

Why Rust?

- **Concurrency Ready**: Future-proof foundation for async file watching
- **Zero-Cost Abstractions**: Raw performance for file operations
- **Portability**: Compile to Windows/macOS/Linux with one codebase

Skill Progression

1. **Task 1**: Rust as backend engine (file I/O)
2. **Task 2**: Rust as system navigator (directory structures)
3. **Task 3**: Rust as creator (file lifecycle management)

"You're not just coding an editor—you're architecting a symphony of system resources."
Ready to think like a full-stack systems engineer? 🦀🔧

Edit this page

Task 1: Core Functionalities

1. Project Setup

- Create Tauri project using `vanilla-ts` template

2. Frontend Structure

- Create HTML with:
  - `<textarea>` for text editing
  - Two buttons (`Open`/`Save`)
- Add CSS for basic layout (optional)

3. Backend Functions (Rust)

- Use **`std::fs`** (built-in Rust module, no external crate needed):
  - `read_file(path: String) -> Result<String,String>`
  - `write_file(path: String, contents: String) -> Result<String, String>`
- Expose these as **Tauri commands** using `#[tauri::command]` and add in `.invoke_handler`

4. Frontend Interaction (TypeScript)

- Use Tauri APIs:
  - **`@tauri-apps/api/plugin-dialog`** for file picker (`open()`, `save()`)
  - **`@tauri-apps/api`** `invoke()` to call Rust commands
- Link button clicks to:
  - Get file path via dialog, (hint: `open()`)
  - Pass path/content between frontend ↔ backend (hint: `invoke()`)

5. Security Configuration, if needed

- In `tauri.conf.json`:
  - Allow **`fs`** access to specific directories
  - Enable **`dialog`** API

Key Technical Requirements

Component	Technology/Package	Purpose
Frontend	Tauri Dialog API	File path selection
Bridge	`invoke()`	TS ↔ Rust communication
Backend	Rust `std::fs`	Read/write files
Error Handling	Rust `Result` type	Propagate I/O errors to frontend

Conclusion Task 1

- Great job! You have done it, now you have a text editor with the core functionalities!
- *"First, make it work. Then make it work harder..."* — Engineers' Mantra

Edit this page

Task 2: File Explorer Sidebar

Build a VS Code-style file hierarchy viewer

1. Frontend Structure Expansion

- Add a sidebar `<div>` next to the textarea
- Create nested HTML lists (`<ul>`, `<li>`) to represent folders/files
- Add folder (📁) and file (📄) icons with CSS

2. Backend Command

- Add a new Rust function:

```
#[tauri::command]
fn get_child_paths(path: String) -> Result<Vec<(String, bool)>, String>
```

  - Uses `std::fs::read_dir` to list directory contents
  - Returns tuples: `(path, is_directory)`
  - Sorts folders first, then files

3. Dynamic Hierarchy Rendering

- Modify "Open File" workflow:
  i. Get file path via `dialog.open()`
  ii. Extract parent directory path
  iii. Call `get_child_paths` to fetch siblings
  iv. Render initial file tree in sidebar

4. Interactive Features

- Add click handlers for:
  - **Folders**: Expand/collapse with ▶/▼ arrows
  - **Files**: Load content into textarea
- Recursive directory loading (fetch children on expand)

5. Security & Error Handling, if needed

- Update `tauri.conf.json` to allow directory traversal:

```
"fs": { "scope": ["$HOME/**"] }
```

- Handle `Result` errors from Rust in TypeScript

Key Technical Requirements

Component	Technology/Package	Purpose
Hierarchy UI	Recursive DOM updates	Dynamic folder expansion
Backend	`std::fs::read_dir`	Directory content listing
State	Event delegation	Handle nested element clicks
Performance	On-demand loading	Only fetch visible directories

Conclusion Task 2

You've transformed the basic editor into a file-centric IDE!
"Complexity is just simplicity with layers of intention."
— *Next: Add a create new file button!* 🚀

Edit this page

Task 3: Add "Create File" Functionality

Extend your editor with file creation capabilities

Core Implementation Roadmap

1. **Backend Command**
    - Add a Rust function `create_file(path: String)` using `std::fs::File::create`
    - Expose via Tauri command
2. **Frontend UI**
    - Add a "📄 New File" button to the toolbar
    - Link it to Tauri's `save()` dialog for path selection
3. **File Explorer Sync**
    - After creation, refresh the parent directory in the sidebar
    - Reuse `get_child_paths` from Task 2
4. **Security, if needed**
    - Allow file creation in permitted directories via `tauri.conf.json`

Final Conclusion

You've achieved to create a text editor with foundational file management capabilities!
"To create is to breathe life into the inert. Now your editor pulses with possibility."
— Next: It seams that you have completed all the challenges for today, What is next?! 🛠️

Edit this page

Next Challenges: Where Will You Take Tauri?

You've built a text editor—a fantastic start! Now, let's dream bigger. Here's how to stretch your skills, fuel your curiosity, and join a community shaping the future of apps.

1. Explore New Horizons

Break boundaries with these ideas:

- **Build tools you wish existed**:
  - A privacy-first note app with local encryption.
  - A markdown-powered journal with cloud sync.
  - A code snippet manager that integrates with GitHub.
- **Solve niche problems**:
  - A minimalist podcast editor for creators.
  - A habit tracker with system-tray reminders.
  - A local-only file organizer with AI tagging.
- **Play with hardware**:
  - A CPU/RAM monitor with real-time graphs.
  - A Bluetooth device configurator for IoT tinkerers.
  - A custom macro pad controller for streamers.

Why?

Tauri lets you blend web creativity with native power. Every project teaches you something new—system APIs, Rust optimizations, or polished UI design.

4. Master the Fundamentals

Skills to quietly level up:

- **Rust's superpowers**: Learn ownership, error handling, and concurrency by optimizing your app's core logic.
- **System integration**: Dive into OS-specific features (menus, notifications, file watchers).
- **Performance tuning**: Profile memory usage, speed up searches, or lazy-load heavy components.

5. Join the Movement

Leave your mark:

- **Contribute to open-source**: Fix a Tauri plugin, improve docs, or share your project template.
- **Build in public**: Post progress on GitHub, write a devlog, or stream your process.
- **Collaborate**: Team up to build a plugin others need (e.g., a calendar picker or terminal emulator).

6. Stay Inspired

Keep the fire alive:

- **Steal ideas**: Rebuild features from apps you admire (e.g., VS Code's extensions, Notion's drag-and-drop).
- **Follow trends**: Experiment with AI integration (Rust + Python?), or build a Tauri-powered PWA.
- **Connect**: Join IP Workshops, and learn from others' "aha!" moments.

Your next step?
Pick *one* idea that makes you think, *"I wanna try that!"*—then start small. The rest will follow. 🚀

"The best way to learn is to build things that excite you." — Someone who probably built a text editor once.

Edit this page