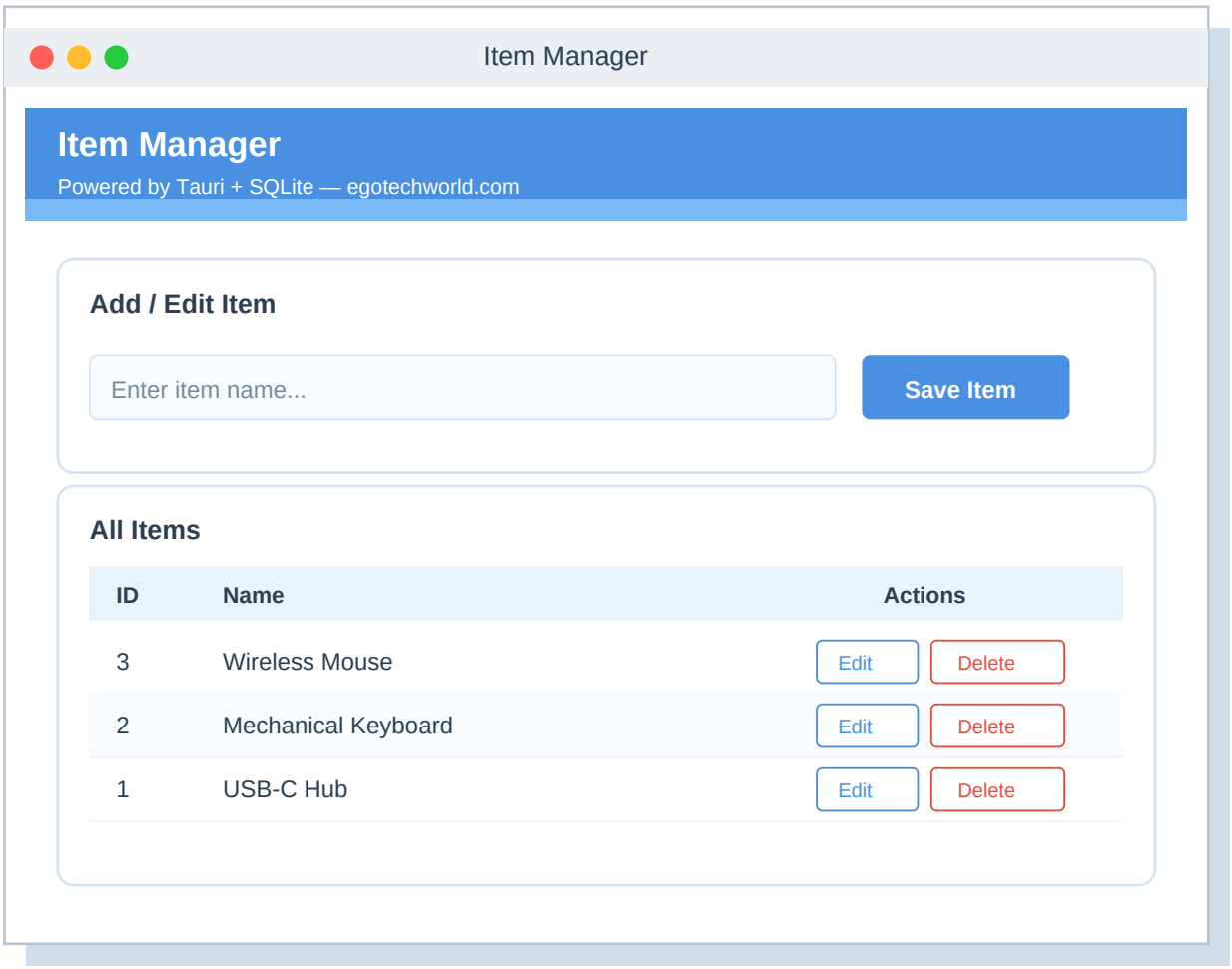
# Build a Windows Desktop CRUD App with Tauri

A Beginner's Step-by-Step Guide

Vanilla JS • Bootstrap 5 • SQLite




*Preview: the finished Item Manager app*

*Published by **egotechworld.com***

*Free Tutorials | Source Code | Developer Tools*

# Table of Contents

# 1. Introduction — Why Tauri?

Tauri is a modern framework for building **tiny, fast, secure desktop applications** using web technologies for the UI and Rust for the backend. Unlike Electron, which bundles a full Chromium browser (resulting in 100+ MB installers), Tauri uses the operating system's native webview. On Windows that means WebView2 — already installed on Windows 10/11.

## Why this stack is great for beginners:

- **Tiny installers** — typically 3 to 10 MB, not 100+ MB.
- **Native performance** — Rust backend is fast and memory-safe.
- **Familiar frontend** — plain HTML, CSS, and JavaScript.
- **Built-in SQLite** — through the official `tauri-plugin-sql`.
- **Secure by default** — explicit permissions for filesystem and database access.

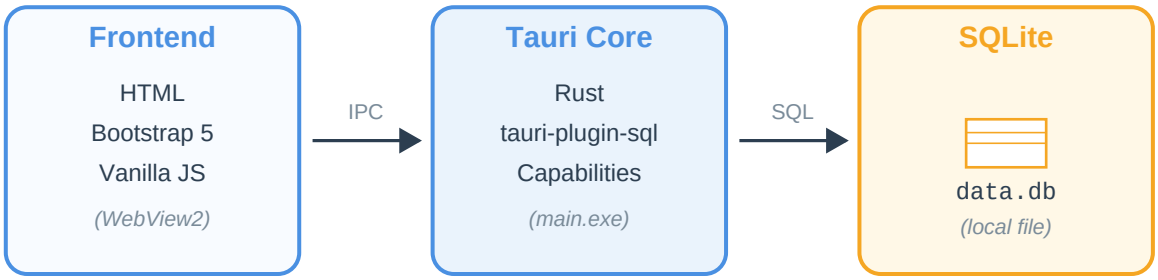

*Figure 1.1 — The 5 high-level steps you'll follow*

**GOOD TO KNOW:** Tauri 2.x is the current major version. This tutorial uses Tauri 2 syntax. If you find Tauri 1 examples online, the plugin setup looks slightly different.

## 2. Architecture Overview

Before writing any code, it helps to picture how the pieces talk to each other. Your app has three layers, all packaged inside a single Windows .exe:

### How the App Works




*All three layers run inside one .exe — no servers, no internet needed.*

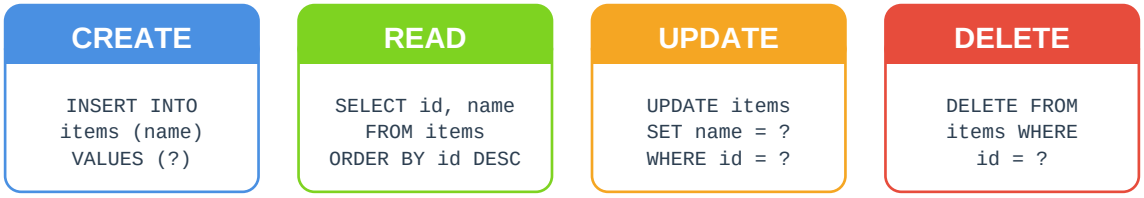
*Figure 2.1 — Frontend ↔ Tauri Core ↔ SQLite database*

The **Frontend** is everything you'd write for a normal web page — HTML, Bootstrap CSS, and Vanilla JavaScript. Inside Tauri it runs in WebView2, the OS-provided browser engine.

The **Tauri Core** is a small Rust program that hosts the WebView and exposes plugins. We register `tauri-plugin-sql` so the JavaScript side can run database queries safely.

**SQLite** is a single-file database. Tauri stores it in the user's AppData folder so it survives between launches.

### CRUD Operations Flow




User clicks button → main.js → db.execute() / db.select() → data.db

*Then loadItems() refreshes the table — UI updates without reload.*

*Figure 2.2 — The four CRUD operations and their SQL*

# 3. Prerequisites

Before we start, you need to install three things on your Windows machine. Each is a one-time setup.

## 3.1 — Microsoft C++ Build Tools

Tauri compiles Rust code, which needs the Microsoft C++ Build Tools. Download **Visual Studio Build Tools** from Microsoft and install the **'Desktop development with C++'** workload.

## 3.2 — Rust

Install Rust from **rustup.rs**. Run the installer, accept defaults, then verify in a new terminal:

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

## 3.3 — Node.js (LTS)

Install **Node.js LTS** from nodejs.org. We need npm to scaffold the Tauri project. Verify with:

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

> **WINDOWS NOTE:** WebView2 Runtime is already installed on Windows 10 and Windows 11. If you are on an older system, download it free from Microsoft.

# 4. Setup Commands — Create the Project

Open **Command Prompt** or **PowerShell**, navigate to the folder where you want your project, and run these commands one by one.

### Step 1 — Scaffold a Tauri Vanilla JS project:

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

When the wizard prompts you, answer like this:

| Question | Answer |
|---|---|
| Project name | item-manager |
| Identifier | com.egotechworld.itemmanager |
| Choose which language for your frontend | JavaScript |
| Choose your package manager | npm |
| Choose your UI template | Vanilla |
| Choose your UI flavor | JavaScript |

### Step 2 — Enter the folder and install dependencies:

```
cd item-manager
npm install
```

### Step 3 — Add the SQL plugin (Rust + JS):

We add the plugin to **both** the Rust backend and the JS frontend.

```
# 1. Add Rust crate (run inside src-tauri folder)
cd src-tauri
cargo add tauri-plugin-sql --features sqlite
cd ..

# 2. Add the JS package (run from project root)
npm install @tauri-apps/plugin-sql
```

### What you should see:

After Step 1, your folder will contain the basic Tauri scaffolding. Step 3 adds two new dependencies — one to Rust's `Cargo.toml` and one to `package.json`.

> **TIP:** If `cargo add` says it's not a recognized command, your Rust install is too old. Run `rustup update` and try again.

# 5. Backend Configuration

Three files need updating to enable the SQLite plugin: `Cargo.toml`, `capabilities/default.json`, and `src-tauri/src/lib.rs`.

## 5.1 — src-tauri/Cargo.toml

If you ran `cargo add` in Step 3, the dependency is already added. Open **src-tauri/Cargo.toml** and verify the `[dependencies]` section contains:

```
[dependencies]
tauri = { version = "2", features = [] }
tauri-plugin-sql = { version = "2", features = ["sqlite"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
```

**TIP:** Exact version numbers may differ — that's fine. What matters is that `tauri-plugin-sql` is present with the `sqlite` feature enabled.

## 5.2 — Permissions (Capabilities)

Tauri 2 uses a **capabilities** system. We need to grant the frontend permission to use the SQL plugin. Open the file:

```
src-tauri/capabilities/default.json
```

Replace the file contents with:

```
{
  "$schema": "../gen/schemas/desktop-schema.json",
  "identifier": "default",
  "description": "Capabilities for the main window",
  "windows": ["main"],
  "permissions": [
    "core:default",
    "sql:default",
    "sql:allow-execute",
    "sql:allow-select",
    "sql:allow-load",
    "sql:allow-close"
  ]
}
```

## Optional — preload the database via tauri.conf.json

You can pre-load the database on startup by adding a `plugins.sql` section at the top level of **src-tauri/tauri.conf.json**:

```
{
  "plugins": {
    "sql": {
      "preload": ["sqlite:data.db"]
    }
  }
}
```

> **TIP:** Add this **plugins** block at the top level, next to `build` and `app`. Don't paste it inside another section.

### 5.3 — src-tauri/src/lib.rs

In Tauri 2, the project usually has a **lib.rs** file holding the `run()` function called from **main.rs**. Register the SQL plugin there:

```
// src-tauri/src/lib.rs
// Register the SQL plugin so the JS frontend can call Database.load()

#[cfg_attr(mobile, tauri::mobile_entry_point)]
pub fn run() {
    tauri::Builder::default()
        // Register the SQL plugin (sqlite feature enabled in Cargo.toml)
        .plugin(tauri_plugin_sql::Builder::default().build())
        .invoke_handler(tauri::generate_handler![])
        .run(tauri::generate_context!())
        .expect("error while running tauri application");
}
```

And **src-tauri/src/main.rs** stays a tiny launcher:

```
// src-tauri/src/main.rs
// Hide the console window on Windows in release builds.
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]

fn main() {
    item_manager_lib::run();
}
```

> **IMPORTANT:** Replace `item_manager_lib` with whatever your library name is — check the `[lib]` section of Cargo.toml. It's usually `app_lib` or your project name with `_lib` appended.

## 6. Frontend UI — index.html

Here's the UI we're aiming for, before we look at the code:

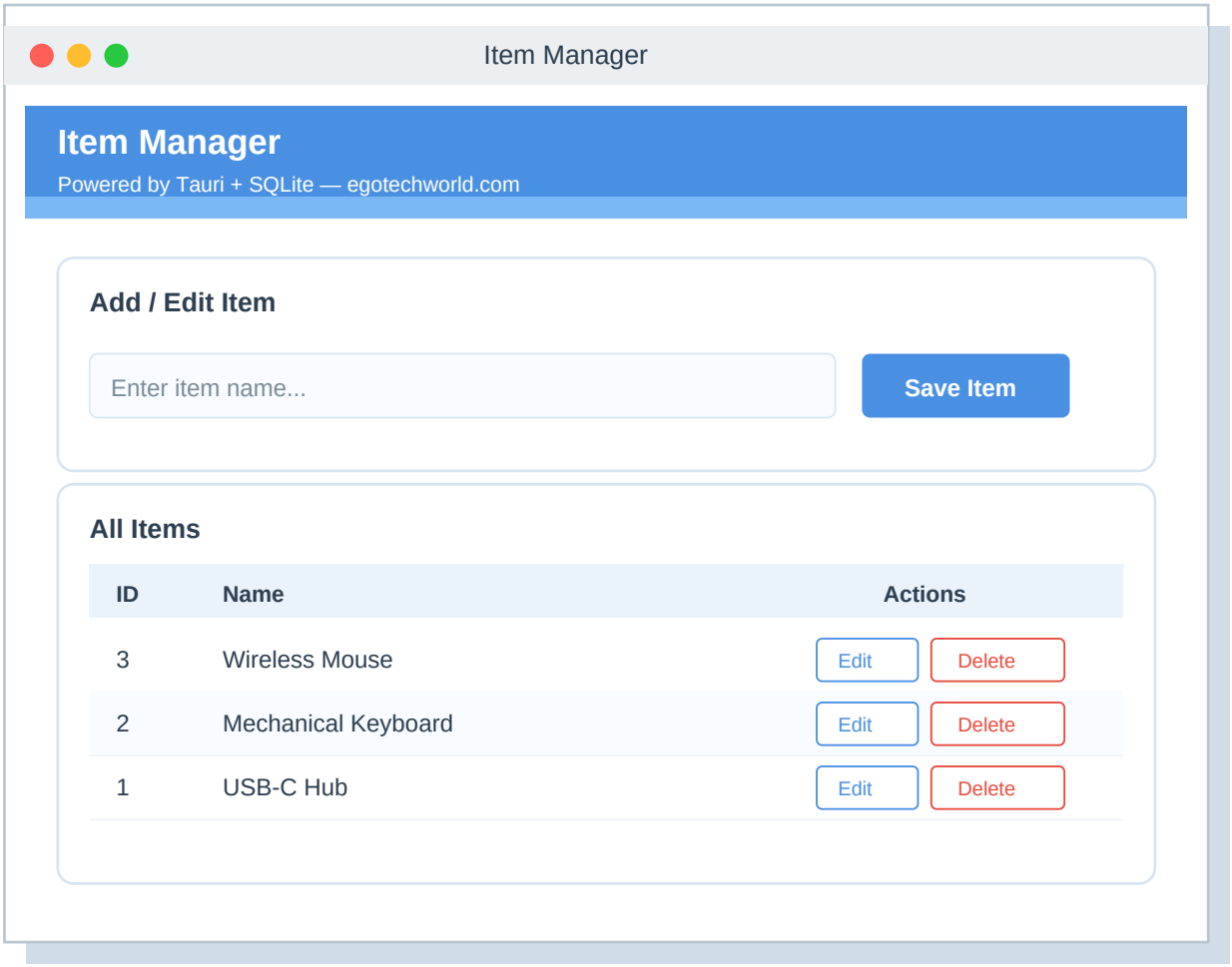



*Figure 6.1 — Final UI: header, form card, items table*

Open **index.html** in the project root and replace its contents with the following. We use Bootstrap 5 from a CDN — no build step.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Item Manager &mdash; egotechworld.com</title>

  <!-- Bootstrap 5 CSS via CDN -->
  <link
    href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
    rel="stylesheet" />

  <style>
    body {
      background: #F8FBFF;
      color: #2C3E50;
      font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", sans-serif;
    }
```

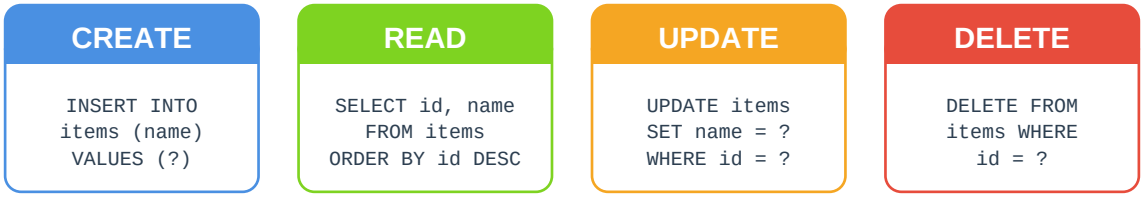
```
    .app-header {
      background: linear-gradient(90deg, #4A90E2, #7AB8F5);
      color: white;
      padding: 18px 24px;
      border-radius: 0 0 12px 12px;
      margin-bottom: 24px;
      box-shadow: 0 2px 8px rgba(74,144,226,0.15);
    }
    .card {
      border: 1px solid #D6E4F2;
      border-radius: 10px;
    }
    .table thead {
      background: #EAF3FC;
      color: #2C3E50;
    }
    .btn-primary {
      background: #4A90E2;
      border-color: #4A90E2;
    }
    .btn-primary:hover {
      background: #357ABD;
      border-color: #357ABD;
    }
    footer {
      color: #7A8B99;
      font-size: 0.85rem;
      text-align: center;
      margin-top: 32px;
      padding: 12px;
    }
  </style>
</head>
<body>

  <!-- Header -->
  <div class="app-header">
    <h3 class="m-0">Item Manager</h3>
    <small>Powered by Tauri + SQLite &mdash; egotechworld.com</small>
  </div>

  <div class="container">
    <!-- Input Form Card -->
    <div class="card mb-4 shadow-sm">
      <div class="card-body">
        <h5 class="card-title">Add / Edit Item</h5>
        <input type="hidden" id="item-id" value="" />

        <div class="row g-2">
          <div class="col-md-9">
            <input type="text" id="item-name" class="form-control"
                   placeholder="Enter item name..." autocomplete="off" />
          </div>
          <div class="col-md-3 d-grid">
            <button id="save-btn" class="btn btn-primary">Save Item</button>
          </div>
        </div>
        <div id="form-msg" class="form-text mt-2"></div>
```

```
      </div>
    </div>

    <!-- Items Table Card -->
    <div class="card shadow-sm">
      <div class="card-body">
        <h5 class="card-title">All Items</h5>
        <div class="table-responsive">
          <table class="table table-hover align-middle">
            <thead>
              <tr>
                <th style="width: 80px;">ID</th>
                <th>Name</th>
                <th style="width: 180px;" class="text-end">Actions</th>
              </tr>
            </thead>
            <tbody id="items-body"></tbody>
          </table>
        </div>
        <div id="empty-msg" class="text-muted text-center py-3 d-none">
          No items yet. Add your first one above!
        </div>
      </div>
    </div>

    <footer>
      &copy; egotechworld.com &mdash; Tauri Desktop Tutorial
    </footer>
  </div>

  <script type="module" src="/main.js"></script>
</body>
</html>
```

## 7. Frontend Logic — main.js

This is the heart of the app. Open **main.js** in the project root and replace everything with the code below. Each function is short and focused.

### CRUD Operations Flow




User clicks button → main.js → db.execute() / db.select() → data.db

*Then loadItems() refreshes the table — UI updates without reload.*

*Figure 7.1 — How each button maps to a SQL operation*

```
// main.js
// Item Manager - CRUD logic using tauri-plugin-sql
// egotechworld.com

import Database from "@tauri-apps/plugin-sql";

// ---- Globals ----
let db = null; // SQLite connection
let editingId = null; // null = creating, number = updating

// ---- DOM references ----
const itemIdInput = document.getElementById("item-id");
const itemNameInput = document.getElementById("item-name");
const saveBtn = document.getElementById("save-btn");
const tbody = document.getElementById("items-body");
const emptyMsg = document.getElementById("empty-msg");
const formMsg = document.getElementById("form-msg");

// ------------------------------------------------------------
// 1) INITIALISE: open DB, create table, then load items
// ------------------------------------------------------------
async function initApp() {
  try {
    db = await Database.load("sqlite:data.db");

    await db.execute(
      `CREATE TABLE IF NOT EXISTS items (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL
       );`
```

```
    );

    await loadItems();
  } catch (err) {
    console.error("DB init failed:", err);
    showMsg("Database error: " + err, true);
  }
}

// ---------------------------------------------------------
// 2) READ: fetch all items and render them in the table
// ---------------------------------------------------------
async function loadItems() {
  const rows = await db.select(
    "SELECT id, name FROM items ORDER BY id DESC;"
  );
  renderRows(rows);
}

function renderRows(rows) {
  tbody.innerHTML = "";

  if (!rows.length) {
    emptyMsg.classList.remove("d-none");
    return;
  }
  emptyMsg.classList.add("d-none");

  for (const row of rows) {
    const tr = document.createElement("tr");
    tr.innerHTML = `
      <td>${row.id}</td>
      <td>${escapeHtml(row.name)}</td>
      <td class="text-end">
        <button class="btn btn-sm btn-outline-primary me-1"
                data-action="edit" data-id="${row.id}">Edit</button>
        <button class="btn btn-sm btn-outline-danger"
                data-action="delete" data-id="${row.id}">Delete</button>
      </td>
    `;
    tbody.appendChild(tr);
  }
}
```

```
// ---------------------------------------------------------
// 3) CREATE / UPDATE: triggered by the Save button
// ---------------------------------------------------------
async function saveItem() {
  const name = itemNameInput.value.trim();
  if (!name) {
    showMsg("Please enter an item name.", true);
    return;
  }

  try {
    if (editingId === null) {
      // CREATE
      await db.execute(
        "INSERT INTO items (name) VALUES ($1);",
        [name]
      );
      showMsg("Item added.", false);
    } else {
      // UPDATE
      await db.execute(
        "UPDATE items SET name = $1 WHERE id = $2;",
        [name, editingId]
      );
      showMsg("Item updated.", false);
    }

    resetForm();
    await loadItems();
  } catch (err) {
    showMsg("Save failed: " + err, true);
  }
}

// ---------------------------------------------------------
// 4) DELETE
// ---------------------------------------------------------
async function deleteItem(id) {
  if (!confirm("Delete this item?")) return;
  try {
    await db.execute("DELETE FROM items WHERE id = $1;", [id]);
    showMsg("Item deleted.", false);
    if (editingId === id) resetForm();
    await loadItems();
  } catch (err) {
    showMsg("Delete failed: " + err, true);
  }
}

// ---------------------------------------------------------
// 5) Switch the form into "edit" mode
// ---------------------------------------------------------
async function startEdit(id) {
  const rows = await db.select(
    "SELECT id, name FROM items WHERE id = $1;",
    [id]
  );
  if (!rows.length) return;
```

```
  editingId = rows[0].id;
  itemIdInput.value = rows[0].id;
  itemNameInput.value = rows[0].name;
  itemNameInput.focus();
  saveBtn.textContent = "Update Item";
}

function resetForm() {
  editingId = null;
  itemIdInput.value = "";
  itemNameInput.value = "";
  saveBtn.textContent = "Save Item";
}
```

```
// ------------------------------------------------------
// 6) Helpers
// ------------------------------------------------------
function showMsg(text, isError) {
  formMsg.textContent = text;
  formMsg.style.color = isError ? "#C0392B" : "#1F8A3B";
  setTimeout(() => (formMsg.textContent = ""), 3000);
}

function escapeHtml(s) {
  return String(s)
    .replace(/&/g, "&amp;").replace(/</g, "&lt;")
    .replace(/>/g, "&gt;").replace(/"/g, "&quot;");
}

// ------------------------------------------------------
// 7) Event wiring
// ------------------------------------------------------
saveBtn.addEventListener("click", saveItem);

itemNameInput.addEventListener("keydown", (e) => {
  if (e.key === "Enter") saveItem();
});

// Event delegation for Edit / Delete buttons in the table
tbody.addEventListener("click", (e) => {
  const btn = e.target.closest("button[data-action]");
  if (!btn) return;
  const id = Number(btn.dataset.id);
  if (btn.dataset.action === "edit") startEdit(id);
  if (btn.dataset.action === "delete") deleteItem(id);
});

// Start the app when the DOM is ready
window.addEventListener("DOMContentLoaded", initApp);
```

**SECURITY:** Notice we use parameterised queries with `$1, $2` instead of string concatenation. This protects against SQL injection — always do this, even in a desktop app.

# 8. Run the App in Development

From the project root, run:

```
npm run tauri dev
```

The first run takes longer because Cargo compiles all Rust dependencies. After that, the desktop window opens with your Item Manager. Try adding a few items, edit one, then close and reopen the app — your data is saved in the local SQLite file.

## Where is data.db stored?

Tauri stores it in the OS app-data folder, not next to the .exe. On Windows the default path looks like this:

```
C:\\Users\\<YourName>\\AppData\\Roaming\\com.egotechworld.itemmanager\\data.db
```

> **TIP:** If you want to inspect the database, install **DB Browser for SQLite** (free) and open the file from that path.

## Common errors and fixes

| Error message | Fix |
|---|---|
| sql:default not allowed | Add the sql:* permissions to capabilities/default.json |
| unresolved import @tauri-apps/plugin-sql | Run: npm install @tauri-apps/plugin-sql |
| the trait bound ... is not satisfied (Rust) | Check Cargo.toml has features = ["sqlite"] |
| link.exe not found | Install Visual Studio Build Tools with C++ workload |

# 9. Build the Windows .exe

When you're happy with the app, compile it into a real Windows installer:

```
npm run tauri build
```

Tauri produces several artefacts in **src-tauri/target/release/**:

- **item-manager.exe** — a portable, single executable.
- **bundle/msi/item-manager_x.x.x_x64_en-US.msi** — Windows MSI installer.
- **bundle/nsis/item-manager_x.x.x_x64-setup.exe** — NSIS setup .exe.

**SIZE COMPARISON:** The bundle output is typically **3 to 10 MB**. Compare that with a similar Electron app at 100+ MB. Share the .msi or setup.exe with friends or upload it to your website.

## Customising the build

Open **src-tauri/tauri.conf.json** to change product name, version, icons, and the window title. Replace icons in **src-tauri/icons/** with your own. The Tauri CLI command `npm run tauri icon path/to/icon.png` will generate every required size automatically from one source image.

# 10. Final Project Structure

Your finished folder layout looks like this:

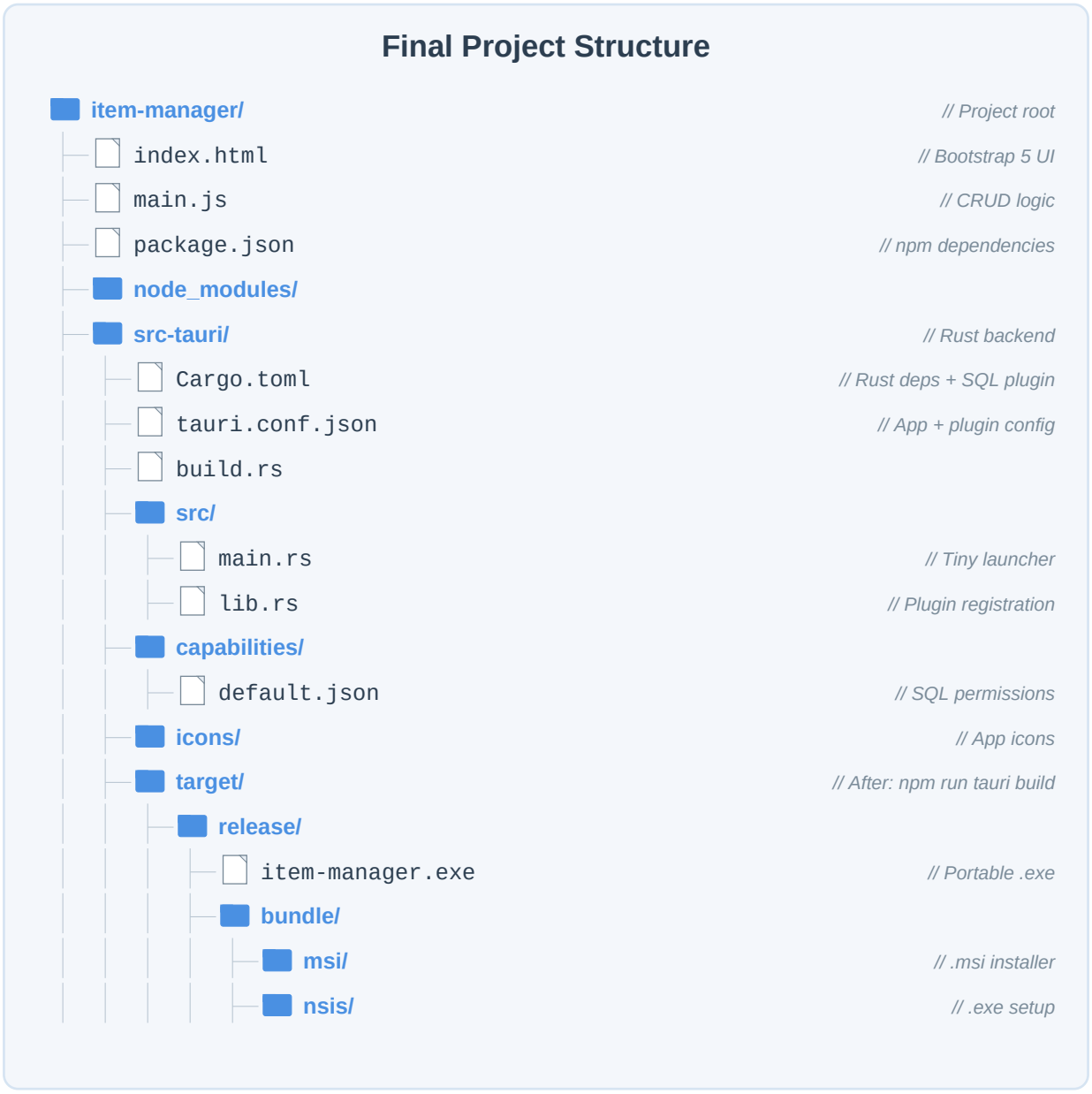



*Figure 10.1 — The complete project tree after building*

## Quick checklist before you ship:

- Replace the default icon with your own brand icon.
- Update `productName` and `version` in tauri.conf.json.
- Test on a clean Windows machine without Rust or Node installed.
- Consider code-signing the .exe for public distribution.

## 11. What's Next?

You now have a working desktop CRUD app. Some natural next steps:

- **Add more columns** — extend the items table with description, price, or category.
- **Search and filter** — add an input that runs `SELECT ... WHERE name LIKE ?`.
- **CSV export** — let users export the table to a .csv file using Tauri's filesystem APIs.
- **Dark mode** — toggle Bootstrap's `data-bs-theme` attribute.
- **Auto-update** — wire up `tauri-plugin-updater` for automatic version updates.
- **Migrations** — use the SQL plugin's migration support for schema changes.



*Happy coding! — egotechworld.com*