**Steps to make Progressive web app using SQLite, html, js, css and node.js**

**Part A- create web app**

sqlite-web-app/

├── /node\_modules/ # Node.js modules

├── /config/ # Config folder for database configuration

│ └── db.js. # Connect to an SQLite database

├── /public/ # Public folder for static assets

│ ├── /css/ # CSS folder

│ │ └── style.css # Main stylesheet

│ └── index.html # Main HTML file

├── /models/ # Sequelize models

│ ├── index.js # Database connection

│ ├── tableA.js # TableA model

│ └── tableB.js # TableB model

├── app.js # Main application file

├── database.sqlite3 # SQLite database file (all tables)

├── package.json # Node.js package config

└── package-lock.json # Auto-generated file

Note: Database.sqlite 3 and package.jason we are going to produce via code. package-lock.json # Auto-generated file

**1. Set Up the Directory**

1. **Create the Project Folder**: open terminal and execute following commands

bash

mkdir sqlite-web-app

cd sqlite-web-app

1. **Initialize a Node.js Project**: Automatically Generate package.json

bash

npm init -y

This will create a package.json with default values.

1. **Install Dependencies**:

After creating package.json, install your required dependencies:  
Install the required packages:

bash

npm install express sequelize sqlite3 body-parser

This will:

1. Add the packages (express, sequelize, sqlite3) to your node\_modules folder.
2. Update the dependencies section of your package.json automatically.

Now, your project is ready to use package.json for managing dependencies and scripts. Let me know if you need help with anything else!

**2. Create the Directory Structure**

mkdir public

mkdir public/css

mkdir models

touch app.js

touch database.sqlite3

touch public/index.html

touch public/css/style.css

touch models/index.js

touch models/tableA.js

touch models/tableB.js

The touch command is a utility in Unix-like operating systems that is primarily used to **create empty files** or update the timestamp of existing files.

**Why Use touch?**

* It’s a quick way to create all necessary files for a project without opening an editor.
* Helps organize the project structure before writing any code.

1. **Open “sqlite-web-app**” **folder in Vs code and Set Up the index.html**

<!-- public/index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta name="theme-color" content="#ffffff"> <!-- Theme color for PWA -->

<link rel="stylesheet" href="/css/style.css"> <!-- Link to CSS -->

<title>SQLite Web App</title>

<link rel="icon" href="/icons/icon-192x192.png"> <!-- Fallback favicon -->

</head>

<body>

<header>

<h1>SQLite Web App</h1>

</header>

<main>

<!-- Table A Form -->

<section>

<h2>Table A</h2>

<form action="/insert-tableA" method="POST">

<label for="name">Name:</label>

<input type="text" id="name" name="name" required>

<button type="submit">Add to Table A</button>

</form>

</section>

<!-- Table B Form -->

<section>

<h2>Table B</h2>

<form action="/insert-tableB" method="POST">

<label for="description">Description:</label>

<input type="text" id="description" name="description" required>

<button type="submit">Add to Table B</button>

</form>

</section>

</main>

<footer>

<p>&copy; 2024 SQLite Web App</p>

</footer>

</body>

</html>

**4. Configure Sequelize and Models**

Write the database connection logic and define models.

**Sequelize** is an Object-Relational Mapping (ORM) library for Node.js which allows you to interact with databases using JavaScript. It supports several databases like PostgreSQL, MySQL, SQLite, and others. Here’s how you can use Sequelize to read and write data in Node.js:

Ref: <https://blog.devgenius.io/connect-node-js-to-sql-database-through-orm-object-relational-mapping-16d323b60423>

**models/index.js**:

const { Sequelize } = require('sequelize');

const sequelize = new Sequelize({

dialect: 'sqlite',

storage: './database.sqlite3',

});

module.exports = sequelize;

**models/tableA.js**:

const { DataTypes } = require('sequelize');

const sequelize = require('./index'); // Import the sequelize instance

const TableA = sequelize.define('TableA', {

id: {

type: DataTypes.INTEGER,

primaryKey: true, // Define as primary key

autoIncrement: true, // Enable auto-increment

allowNull: false, // Ensure it can't be null

},

name: {

type: DataTypes.STRING,

allowNull: false, // Ensure it can't be null

},

}, {

timestamps: true, // Includes createdAt and updatedAt

});

module.exports = TableA;

**models/tableB.js**:

const { DataTypes } = require('sequelize');

const sequelize = require('./index'); // Import the sequelize instance

const TableB = sequelize.define('TableB', {

id: {

type: DataTypes.INTEGER,

primaryKey: true, // Define as primary key

autoIncrement: true, // Enable auto-increment

allowNull: false, // Ensure it can't be null

},

description: {

type: DataTypes.STRING,

allowNull: false, // Ensure it can't be null

},

}, {

timestamps: true, // Includes createdAt and updatedAt

});

module.exports = TableB;

1. **Write the Main Application Logic**

**app.js**

**Set Up a Simple Node.js Server**

If you are using Node.js with Express, you can set up a simple HTTP server to serve your app and enable PWA features.

This Express server:

* Serves static files from the public directory (where your PWA assets like index.html, manifest.json, service-worker.js, and CSS are located).
* Serves your main index.html when you visit the root URL (/).
* Listens on port 3000 (or any environment-specified port).

const express = require('express');

const path = require('path');

const { Sequelize, DataTypes } = require('sequelize');

const app = express(); // Initialize the app

// Middleware to parse JSON and URL-encoded data

app.use(express.json());

app.use(express.urlencoded({ extended: true }));

// Serve static files from the 'public' directory

app.use(express.static(path.join(\_\_dirname, 'public')));

// Database setup

const sequelize = new Sequelize({

dialect: 'sqlite',

storage: './database.sqlite3',

});

// Define models

const TableA = sequelize.define('TableAs', {

id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },

name: { type: DataTypes.STRING, allowNull: false },

});

const TableB = sequelize.define('TableBs', {

id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },

description: { type: DataTypes.STRING, allowNull: false },

});

// Sync database

sequelize.sync().then(() => {

console.log('Database synchronized');

});

// Routes for inserting data

app.post('/insert-tableA', async (req, res) => {

try {

const { name } = req.body;

await TableA.create({ name });

res.send('Data inserted into Table A');

} catch (error) {

res.status(500).send('Error inserting data into Table A');

}

});

app.post('/insert-tableB', async (req, res) => {

try {

const { description } = req.body;

await TableB.create({ description });

res.send('Data inserted into Table B');

} catch (error) {

res.status(500).send('Error inserting data into Table B');

}

});

// Start the server

const PORT = 3000;

app.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});

1. **Add Styling**

Add styles in public/css/style.css:

body {

font-family: Arial, sans-serif;

padding: 20px;

}

h1 {

text-align: center;

}

form {

margin-bottom: 20px;

}

label, input {

margin-right: 10px;

}

table {

width: 100%;

border-collapse: collapse;

}

table, th, td {

border: 1px solid #ddd;

}

th, td {

padding: 8px;

text-align: left;

}

1. **Test and Run the Application**
2. **Navigate to Your Project Directory**  
   Open your terminal or command prompt, then navigate to the root of your project:

bash

cd /path/to/sqlite-web-app

1. **Run npm init or npm init -y**

2.1) To create package.json interactively (it will ask for project details):

bash

npm init

2.2) To create it with default values (quicker):

bash

npm init -y

1. **Install Required Dependencies**  
   After initializing, install the necessary dependencies (express, sequelize, and sqlite3):

bash

npm install express sequelize sqlite3

This will add a dependencies section in your package.json listing these packages.

1. **Start the Server**:

bash

node app.js

1. **Open the App**:  
   Visit http://localhost:3000 in your browser.
2. **Test Insertions**:  
   Add data to **Table A** and **Table B** through the forms.
3. **Fetch Data**:  
   You can test the /fetch-data endpoint using a browser or a tool like Postman.

**Part B- Convert web app to PWA**

**Steps to Make Your App a PWA**

sqlite-web-app/

├── /node\_modules/ # Node.js modules

├── /config/ # Config folder for database configuration

│ └── db.js. # Connect to an SQLite database

├── /public/ # Public folder for static assets

│ ├── /css/ # CSS folder

│ │ └── style.css # Main stylesheet

│ ├── /icons/ # Icons folder

│ │ ├── icon-192x192.png # 192x192 icon

│ │ └── icon-512x512.png # 512x512 icon

│ ├── index.html # Main HTML file

│ ├── manifest.json # PWA manifest

│ └── sw.js # Service Worker file

├── /models/ # Sequelize models

│ ├── index.js # Database connection

│ ├── tableA.js # TableA model

│ └── tableB.js # TableB model

├── app.js # Main application file

├── database.sqlite3 # SQLite database file

├── package.json # Node.js package config

└── package-lock.json # Auto-generated file

1. **Add a Web App Manifest**

* Create a manifest.json file in the public folder to define your app's metadata.
* The manifest.json file is the only file that every extension using WebExtension APIs must contain.
* Using manifest.json, you specify basic metadata about your extension such as the name and version, and can also specify aspects of your extension's functionality (such as background scripts, content scripts, and browser actions).
* It is a [JSON](https://developer.mozilla.org/en-US/docs/Glossary/JSON)-formatted file, with one exception: it is allowed to contain "//"-style comments.

Check this [link](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json) to learn more about manifest.json

Example manifest.json:

{

"name": "SQLite Web App",

"short\_name": "SQLiteApp",

"description": "A web app to manage SQLite data.",

"start\_url": "/",

"display": "standalone",

"background\_color": "#ffffff",

"theme\_color": "#0078d7",

"icons": [

{

"src": "/icons/icon-192x192.png",

"type": "image/png",

"sizes": "192x192"

},

{

"src": "/icons/icon-512x512.png",

"type": "image/png",

"sizes": "512x512"

}

]

}

1. **Add App Icons**

* Place app icons in a folder like public/icons/.
* Ensure the icons match the sizes defined in manifest.json.

1. **Service Worker for Offline Capability**

* Create a service worker file public/sw.js to handle caching.

Example sw.js:

//this is for PWA

self.addEventListener('install', (event) => {

console.log('Service Worker: Installed');

event.waitUntil(

caches.open('static-cache').then((cache) => {

console.log('Service Worker: Caching App Shell');

return cache.addAll([

'/',

'/css/style.css',

'/manifest.json',

'/icons/icon-192x192.png',

'/icons/icon-512x512.png',

]);

})

);

});

self.addEventListener('fetch', (event) => {

event.respondWith(

caches.match(event.request).then((response) => {

return response || fetch(event.request);

})

);

});

1. **Update index.html**

**Ensure Service Worker and Manifest are Configured Correctly**

Ensure that your service-worker.js is properly registered in your HTML or main JavaScript file, like index.html or app.js.

In the index.html, make sure the following is added inside the <head> tag:

html

<link rel="manifest" href="/manifest.json"> This links the manifest.json file to your web app.

 Link the manifest and register the service worker.

 Modify <head> in index.html:

<link rel="manifest" href="/manifest.json"> <!-- Link to Web App Manifest -->

<!-- PWA: Service Worker Registration -->

<script>

if ('serviceWorker' in navigator) {

navigator.serviceWorker.register('/sw.js')

.then(() => console.log("Service Worker registered successfully"))

.catch((error) => console.error("Service Worker registration failed:", error));

}

</script>

1. **Update app.js – add following code at the bottom**

in your app.js or a separate JavaScript file, register the service worker:

//Add the route to serve the sw.js file.

//this is for PWA

app.get('/sw.js', (req, res) => {

res.setHeader('Content-Type', 'application/javascript');

res.sendFile(path.join(\_\_dirname, 'public', 'sw.js'));

});

1. **Make sure icon images are inside icons folder**

=========================================

Steps to Run:

1. Start your app again:

bash

node app.js

1. Open [http://localhost:3000](http://localhost:3000/) in your browser.

==========================================

To run your Node.js-based Progressive Web App (PWA), follow these steps:

1. **Install Dependencies**

Before running the app, ensure that all dependencies are installed. Navigate to your project directory and run:

bash

npm install

This will install the required dependencies from your package.json file (like Express and other Node.js modules).

1. **Start the Server**

* Open terminal
* Go to your directory (cd sqlite\_web\_app)

Run your server using Node.js. In the terminal, navigate to your project folder (if not already there) and run:

bash

node app.js

This should start your server, and you can access your PWA by going to http://localhost:3000 in your browser.

1. **Test PWA Features**

Right click on you app web page and choose **“Inspect” to open DevTools**

A screenshot of a phone

Description automatically generated

To test your PWA:

* **Manifest**: Check if the manifest file is correctly linked by inspecting the page source or using Chrome DevTools under the "Application" tab.
* **Service Worker**: In Chrome, go to DevTools → "Application" → "Service Workers" to confirm that the service worker is registered and working.
* **Offline Testing**: In DevTools, under "Network", you can toggle "Offline" mode to simulate how the app behaves without an internet connection (it should serve cached files).

To confirm that your service worker is registered and working using Chrome's DevTools, follow these steps:

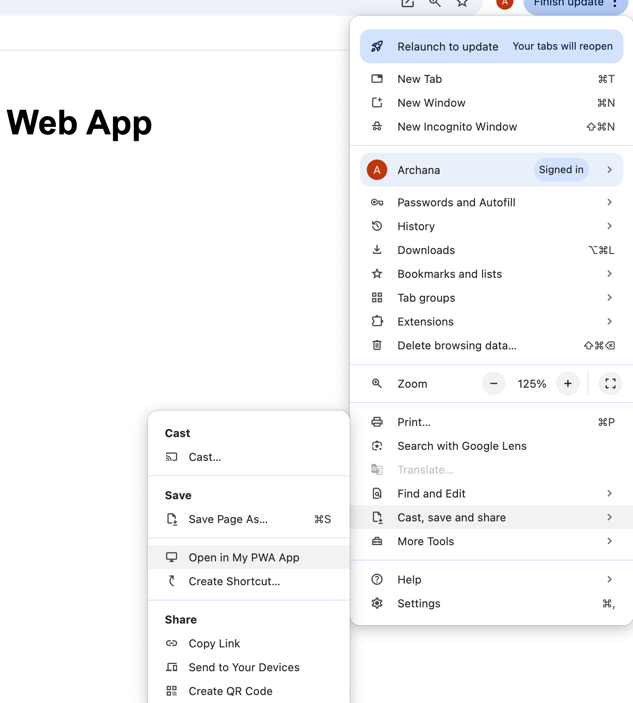
**Ensure HTTPS Connection**: Your PWA must be served over HTTPS (except on localhost for development). If you're testing locally, you can use localhost or set up HTTPS for your server.

1. **Open Your PWA in Chrome**

* Make sure your PWA is accessible from a URL in Chrome.
* Open the PWA's URL in Google Chrome.

1. **Install the PWA in Chrome**

* Click on 3 dot
* Go to cast,, save and share
* Click “Install my PWA app” (Screen short is showing open in my PWA app because mine is already installed)

****

1. **Access the Installed PWA**

After installation, your PWA will appear in your **Applications** folder and **Launchpad**. You can open it like any other application on your MacBook.

1. **Verify Installation**

To verify that your app is installed as a PWA:

* Open the **Applications** folder or open **launchpad** and look for your app.
* You can pin the app to the **Dock** for easy access.
* It should run in a standalone window without the browser UI (i.e., no address bar).