# **Bringing SQLite to the Browser: A Journey with WebAssembly**

By Julian Wilkison-Duran and ChatGPT

In today’s development world, it’s becoming increasingly common to handle complex operations directly in the browser. With WebAssembly (WASM) leading this revolution, one powerful use case is running SQLite databases inside the browser. This post walks through a real-world experience integrating SQLite into a browser-based environment using WebAssembly, highlighting the challenges encountered and how they were overcome.

## **Project Overview**

The goal of this project was to import an existing SQLite database into the browser using WebAssembly. With the SQLite Wasm (WebAssembly) module as the core, we aimed to fetch a pre-existing SQLite database, deserialize it, and interact with the data directly in the browser.

### **Challenges and Initial Attempts**

#### **Starting with Documentation and Fetching the Database**

We began by reviewing the official SQLite Wasm documentation to understand the steps necessary to load a database into the browser. A promising solution was found in [SQLite's Wasm cookbook](https://sqlite.org/wasm/doc/trunk/cookbook.md), specifically step 2.2, which describes fetching a database using fetch() and handling it as a binary file.

The initial approach involved fetching the database and using sqlite3.Database() to open the fetched database file. Here is where we hit the first hurdle: **"sqlite3.Database() is not a constructor"**.

At this point, I guided you through understanding that the Wasm environment needs specific steps to properly initialize, and the problem stemmed from incorrectly attempting to create an SQLite database instance before the module was fully loaded.

### **The Evolution of the Solution**

#### **Step 1: Understanding the SQLite Wasm Module Initialization**

The first challenge was getting the SQLite Wasm module (sqlite3.wasm and sqlite3.js) to load properly. We experimented with various script tags to ensure the module would load and initialize before attempting any database interaction.

The following script block turned out to be critical for initializing the module correctly:

if(globalThis.window!==globalThis){

let sqlite3Js = 'sqlite3.js';

const urlParams = new URL(globalThis.location.href).searchParams;

if(urlParams.has('sqlite3.dir')){

sqlite3Js = urlParams.get('sqlite3.dir') + "/" + sqlite3Js;

}

importScripts(sqlite3Js);

}

globalThis.sqlite3InitModule().then(function(sqlite3){

//sqlite3 can be used here to load the db

});



This step was crucial because it ensured that the Wasm module was properly loaded in either the main thread or a worker thread. The use of importScripts() for workers and the careful handling of URL parameters helped ensure the paths to the necessary Wasm and JS files were correct.

#### **Step 2: Resolving the Wasm Module Fetch Issue**

Despite initializing the module correctly, you faced another issue: **sqlite3.wasm wasn’t showing up in the network tab**, even though sqlite3.js was loading. This suggested that the .wasm file wasn’t being fetched or located correctly.

Through further debugging, you realized that by changing the script source to a locally served version (e.g., http://localhost:8000/wasm/sqlite3.js), sqlite3.wasm could be correctly downloaded. This breakthrough made it possible to finally fetch both the .js and .wasm files needed for the module to run.

#### **Step 3: Fetching and Deserializing the SQLite Database**

Once the module was loaded, we worked on fetching and deserializing the SQLite database. The key was to use the fetch() API to download the SQLite file as an ArrayBuffer, and then load it into SQLite using the sqlite3.capi.sqlite3\_deserialize() function. Here’s the code that made it work:

const db = new oo.DB();

await fetch('http://localhost:8000/testSuiteDB.sqlite')

.then(response => response.arrayBuffer())

.then(arrayBuffer => {

const bytes = new Uint8Array(arrayBuffer);

const p = sqlite3.wasm.allocFromTypedArray(bytes);

db.onclose = { after: function(){ sqlite3.wasm.dealloc(p); } };

const rc = capi.sqlite3\_deserialize(

db.pointer, 'main', p, bytes.length, bytes.length, 0

);

db.checkRc(rc);

});



In this final solution, sqlite3.wasm.allocFromTypedArray() was used to allocate memory on the Wasm heap and deserialize the database with sqlite3\_deserialize(). The database was then available to execute SQL queries directly in the browser.

### **Overcoming Technical Hurdles**

One of the key takeaways from this journey is that WebAssembly, while powerful, requires careful attention to initialization and memory management, particularly when deserializing data like an SQLite database. Here's a summary of the key challenges we faced:

* **Module Initialization**: Ensuring that sqlite3.wasm and sqlite3.js were correctly located and loaded was vital. We learned that the Wasm files need to be carefully managed, especially in worker threads, where importScripts() plays a critical role.
* **Fetching the Database**: Using fetch() to download a database as an ArrayBuffer and converting it into a Uint8Array was a subtle but essential step. Handling binary data in JavaScript required an understanding of how WebAssembly allocates and deallocates memory.
* **Deserialization**: The sqlite3\_deserialize() function proved crucial in importing the database. Its role in loading binary SQLite data into a Wasm-allocated memory space was a key discovery that ultimately unlocked the full functionality of SQLite in the browser.

### **Final Results: Running SQLite in the Browser**

With the successful deserialization of the SQLite database, you were able to query it directly in the browser. The final code not only fetches the SQLite database but also allows for executing SQL queries such as SELECT DISTINCT name FROM tag:

db.exec({

sql: "SELECT DISTINCT name FROM tag",

rowMode: 'array',

callback: function(row){

log("row ",++this.counter,"=",row);

}.bind({counter: 0})

});



This query demonstrates the power of SQLite Wasm: you can run complex SQL commands and interact with your data entirely within the browser without needing any server-side interaction.

### **Lessons Learned**

This project showcased the growing capability of WebAssembly to bring traditionally server-side operations into the client. Here are a few lessons from the journey:

1. **Module Initialization**: Carefully managing how WebAssembly modules load is critical. Pay close attention to paths, especially when serving from local servers or running in worker threads.
2. **Binary Data Management**: Handling databases in a browser environment requires an understanding of how to manage binary data, particularly when it comes to WebAssembly’s memory heap.
3. **Documentation**: While official documentation is helpful, it often assumes familiarity with concepts like memory allocation and binary deserialization. Sometimes, deeper investigation and experimentation are necessary to truly understand how things fit together.
4. **Debugging WebAssembly**: The debugging process for WebAssembly modules can be tricky. Keeping a close eye on the network tab, ensuring proper module loading, and adding lots of logging can help pinpoint where things go wrong.

### **Conclusion**

Thanks to the power of WebAssembly and SQLite, it's possible to build rich, interactive applications that run entirely within the browser, including complex database operations. The project highlighted how important it is to manage module initialization and memory in the WebAssembly world, and the result is a fully functional SQLite instance running inside the browser.

The journey wasn’t without its bumps, but by systematically addressing each issue—from module loading to database deserialization—we were able to achieve the goal. If you're looking to bring SQLite databases into the browser, the steps outlined here should help get you started on the right path.

This project was a fantastic example of how emerging technologies like WebAssembly can transform the web development landscape. The potential is vast, and it’s exciting to imagine what other traditionally server-side operations can be brought into the browser in the near future.