# 🗃️ Choosing a Storage Solution in the Browser

When building rich web applications — especially those with complex, structured data — developers typically choose between **two main browser-native storage options**: **IndexedDB** and **OPFS (Origin Private File System)**.

## 🔹 Option 1: IndexedDB

IndexedDB is a low-level, asynchronous key-value store built into all major browsers. It’s well-supported, powerful, and designed for structured storage.

**✅ Benefits:**

* Built-in browser support (no extra permissions or headers required)
* Works offline
* Can store complex objects (via structured cloning)
* Large storage limits (hundreds of MBs)

**❌ Shortcomings:**

* Poor developer ergonomics
* No relational data model — queries are manual and index-driven
* Hard to migrate or manage schemas
* Difficult to debug or visualize
* Verbose and repetitive APIs

## 🔹 Option 2: OPFS (Origin Private File System)

OPFS is part of the modern [File System Access API](https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API), which allows websites to interact with a real (sandboxed) file system — enabling powerful storage options like embedded databases (e.g., SQLite).

**✅ Benefits:**

* Persistent, file-based storage
* Enables use of SQLite directly in the browser
* Ideal for apps needing relational data
* More powerful than key-value stores for complex relationships

**❌ Shortcomings:**

* Requires **Cross-Origin Isolation** (COOP/COEP HTTP headers)
* Not universally supported (some browser limitations)
* Not visible in DevTools (can make debugging trickier)
* Slightly more complex setup (especially when combined with WebAssembly)

## ✅ Our Choice: OPFS

We chose **OPFS** for our project because it provided us with **true relational data support in the browser**, persistence between sessions, and compatibility with **SQLite** — the gold standard for local structured databases.

For a game application with interconnected data like players, game sessions, and score history, **a relational database was the most natural fit**.

## 🚀 Enhancing Developer Experience with SQLocal + Drizzle

To accelerate development and improve safety and maintainability, we used [**SQLocal**](https://sqlocal.dallashoffman.com/) — a lightweight library that wraps SQLite running in WebAssembly with a smooth developer experience for browser environments.

Even better, SQLocal comes with built-in support for **Drizzle ORM**, a modern, type-safe SQL query builder designed for DX. By using Drizzle:

* We defined our schemas directly in TypeScript
* We generated migration SQL files
* We avoided writing repetitive queries by hand
* We got full IntelliSense and compile-time safety

This drastically improved both the **reliability** and **velocity** of our frontend data layer.

## 🧩 Challenges We Faced

While the end result was powerful, the path wasn’t without obstacles:

### 1. Understanding OPFS Requirements

* We had to enable **Cross-Origin Isolation**, which meant configuring Vite to serve Cross-Origin-Opener-Policy and Cross-Origin-Embedder-Policy headers.
* Until those headers were in place, SQLite fell back to **in-memory mode**, causing silent data loss on refresh.

### 2. No exec() API in SQLocal

* SQLocal uses **tagged template strings** (like sql`...`), which made it impossible to run multi-line SQL migration scripts directly from a string.
* The batch() method was also insufficient in our case.
* We solved this by \*\*manually converting SQL migration files into explicit await sql\…`statements\*\*, wrapping them in acreateTables()` function. It wasn’t pretty, but it was reliable and predictable.

### 3. Drizzle doesn’t auto-create tables

* Drizzle provides schema typing, but doesn’t create tables itself.
* We used Drizzle’s CLI to generate .sql migration files and then executed those via createTables() using raw SQL in the browser.

### 4. Debugging Browser-Side SQLite

* OPFS isn’t visible in DevTools, which made initial debugging more challenging.
* We relied heavily on console logs and test queries to confirm state.
* Importantly:  
  **persisted: false does not mean the data will be lost** when the tab or browser closes.  
  It only means the browser **may** clear the storage under memory pressure.  
  In practice, OPFS will still survive across sessions unless manually cleared or evicted by the browser.

## 🏁 Conclusion

Despite the early hurdles, our final setup using **OPFS + SQLocal + Drizzle** gives us:

* **Persistent, relational storage in the browser**
* **Type-safe querying and schema evolution**
* **A scalable foundation for complex data-heavy PWAs**

It’s fast, robust, and developer-friendly — everything we need to build a high-quality browser-first game with modern tooling.