Best Practices for Using IndexedDB

- Keeping your app predictable
  - Not everything can be stored in IndexedDB on all platforms
  - Writing to storage may fail
  - Stored data may have been modified or deleted by the user
  - Stored data may be out of date
- Keeping your app performant
- Conclusions

When a user first loads a website or application, there's often a fair amount of work involved in constructing the initial application state that's used to render the UI. For example, sometimes the app needs to authenticate the user client-side and then make several API requests before it has all the data it needs to display on the page.

Storing the application state in IndexedDB can be a great way to speed up the load time for repeat visits. The app can then sync up with any API services in the background and update the UI with new data lazily, employing a stale-while- revalidate strategy.

Another good use for IndexedDB is to store user-generated content, either as a temporary store before it is uploaded to the server or as a client-side cache of remote data - or, of course, both.

However, when using IndexedDB there are many important things to consider that may not be immediately obvious to developers new to the APIs. This article answers common questions and discusses some of the most important things to keep in mind when persisting data in IndexedDB.

Keeping your app predictable

A lot of the complexities around IndexedDB stem from the fact that there are so many factors you (the developer) have no control over. This section explores many of the issues you must keep in mind when working with IndexedDB.

Not everything can be stored in IndexedDB on all platforms

If you are storing large, user-generated files such as images or videos, then you may try to store them as `File` or `Blob` objects. This will work on some platforms but fail on others. Safari on iOS, in particular, cannot store `Blob`s in IndexedDB.

Luckily it is not too difficult to convert a `Blob` into an `ArrayBuffer`, and vice versa. Storing `ArrayBuffer`s in IndexedDB is very well supported.

Remember, however, that a `Blob` has a MIME type while an `ArrayBuffer` does not. You will need to store the type alongside the buffer in order to do the conversion correctly.

To convert an `ArrayBuffer` to a `Blob` you simply use the `Blob` constructor.

```
function arrayBufferToBlob(buffer, type) {
  return new Blob([buffer], { type: type });
}
```

The other direction is slightly more involved, and is an asynchronous process. You can use a `FileReader` object to read the blob as an `ArrayBuffer`. When the reading is finished a `loadend` event is triggered on the reader. You can wrap this process in a `Promise` like so:

```
function blobToArrayBuffer(blob) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.addEventListener('loadend', () => {
      resolve(reader.result);
    });
    reader.addEventListener('error', reject);
    reader.readAsArrayBuffer(blob);
  });
}
```

Writing to storage may fail

Errors when writing to IndexedDB can happen for a variety of reasons, and in some cases these reasons are outside of your control as a developer. For example, some browsers currently don't allow writing to IndexedDB when in private browsing mode. There's also the possibility that a user is on a device that's almost out of disk space, and the browser will restrict you from storing

anything at all.

New storage APIs are currently being developed that will allow developers to get an estimate of how much storage space is available prior to writing as well as request larger storage quota or even persistent storage, meaning the user can opt-in to preserving data from some sites even when performing standard clear cache/cookies operations.

Because of this, it's critically important that you always implement proper error handling in your IndexedDB code. This also means it's generally a good idea to keep application state in memory (in addition to storing it), so the UI doesn't break when running in private browsing mode or when storage space isn't available (even if some of the other app features that require storage won't work).

You can catch errors in IndexedDB operations by adding an event handler for the `error` event whenever you create an `IDBDatabase`, `IDBTransaction` or `IDBRequest` object.

```
const request = db.open('example-db', 1);
request.addEventListener('error', (event) => {
  console.log('Request error:', request.error);
};
```

Stored data may have been modified or deleted by the user

Unlike server-side databases where you can restrict unauthorized access, client-side databases are accessible to browser extensions and developer tools, and they can be cleared by the user.

While it may be uncommon for users to modify their locally stored data, it's quite common for users to clear it. It's important that your application can handle both of these cases without erroring.

Stored data may be out of date

Similar to the previous section, even if the user hasn't modified the data themselves, it's also possible that the data they have in storage was written by an old version of your code, possibly a version with bugs in it.

IndexedDB has built-in support for schema versions and upgrading via its `IDBOpenDBRequest.onupgradeneeded()` method; however, you still need to write your upgrade code in such a way that it can handle the user coming from a previous version (including a version with a bug).

Unit tests can be very helpful here, as it's often not feasible to manually test all possible upgrade paths and cases.

Keeping your app performant

One of the key features of IndexedDB is its asynchronous API, but don't let that fool you into thinking you don't need to worry about performance when using it. There are a number of cases where improper usage can still block the main thread, which can lead to jank and unresponsiveness.

As a general rule, reads and writes to IndexedDB should not be larger than required for the data being accessed.

While IndexedDB makes is possible to store large, nested objects as a single record (and doing so is admittedly quite convenient from a developer perspective), this practice should be avoided. The reason is because when IndexedDB stores an object, it must first create a structured clone of that object, and the structured cloning process happens on the main thread. The larger the object, the longer the blocking time will be.

This presents some challenges when planning how to persist application state to IndexedDB, as most of the popular state management libraries (like Redux) work by managing your entire state tree as a single JavaScript object.

While managing state in this way has many benefits (e.g. it makes your code easy to reason about and debug), and while simply storing the entire state tree as a single record in IndexedDB may be tempting and convenient, doing this after every change (even if throttled/debounced) will result in unnecessary blocking of the main thread, it will increase the likelihood of write errors, and in some cases it will even cause the browser tab to crash or become unresponsive.

Instead of storing the entire state tree in a single record, you should break it up into individual records and only update the records that actually change.

The same is true if you store large items like images, music, or video in IndexedDB. Store each item with its own key rather than inside a larger object, so that you can retrieve the structured data without paying the cost of also retrieving the binary file.

As with most best practices, this is not an all-or-nothing rule. In cases where it's not feasible to break up a state object and just write the minimal change-set, breaking up the data into sub-trees and only writing those is still preferable to always writing the entire state tree. Little improvements are better than no improvements at all.

Lastly, you should always be measuring the performance impact of the code you write. While it's true that small writes to IndexedDB will perform better than large writes, this only matters if the

writes to IndexedDB that your application is doing actually lead to long tasks that block the main thread and degrade the user experience. It's important to measure so you understand what you're optimizing for.

Conclusions

Developers can leverage client storage mechanisms like IndexedDB to improve the user experience of their application by not only persisting state across sessions but also by decreasing the time it takes to load the initial state on repeat visits.

While using IndexedDB properly can dramatically improve user experience, using it incorrectly or failing to handle error cases can lead to broken apps and unhappy users.

Since client storage involves many factors outside of your control, it's critical your code is well tested and properly handles errors, even those that may initially seem unlikely to occur.

□□□ #3
□ □□□ □□□ 2026-10-31 21:58:07 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:19 UTC