# Browser Storage Course

Flavio Copes

Updated August 3, 2026

## Contents



Build reliable browser state with cookies, Web Storage, IndexedDB, Cache Storage, quotas, persistence, and origin-private files.

**What you'll learn:** Build and test a local-first field-notes app with deliberate storage ownership, safe upgrades, offline resources, export, cleanup, and recovery.

## Choose and observe storage

Map each value's owner and lifetime, understand browser boundaries, and inspect existing state.

### Follow one piece of data

Start with ownership, lifetime, size, access, and recovery instead of choosing a browser API by habit.

Browser storage is not one box. Cookies, Web Storage, IndexedDB, Cache Storage, and files solve different jobs.

Before choosing an API, follow one value through its complete life. Ask who creates it, which code reads it, whether the server needs it, how large it can grow, when it expires, and how the user recovers it after loss.

In our field-notes app, a color theme is a small replaceable preference. An unfinished editor draft belongs to one tab. Saved notes are structured records. Help pages are fetched responses. Attachments are files. Those are five different storage shapes.

My advice is to assume browser data can disappear. A user can clear site data, private sessions end, devices fail, and browsers may evict best-effort storage. If losing a value would be serious, provide export or server synchronization.

Create a storage inventory for theme, draft, saved note, cached help page, attachment, and login session. Record owner, reader, lifetime, maximum size, cleanup rule, and recovery path for each.

### Understand origin and site boundaries

Separate origins, sites, tabs, and top-level storage partitions so data does not cross a boundary by accident.

A storage boundary begins with the URL. Small URL changes can create a different browser bucket.

An origin is the scheme, host, and port. `https://notes.test` and `http://notes.test` have different origins. So do `notes.test` and `app.notes.test`. Web Storage and IndexedDB normally follow this origin boundary.

Cookies use domain and path matching, while `SameSite` reasons about sites rather than origins. Embedded third-party content may also receive partitioned storage keyed by the top-level site. Do not treat these terms as interchangeable.

`sessionStorage` adds another boundary: the top-level tab. Two same-origin tabs share `localStorage`, but normally have separate `sessionStorage` page sessions. We will use that difference for editor drafts.

Open the same test page on HTTP, HTTPS, a subdomain, another port, and a second tab. Predict which values are shared before inspecting each storage area in DevTools.

### Choose storage from the job

Match cookies, Web Storage, IndexedDB, Cache Storage, and OPFS to the shape and destination of the data.

Choose the smallest API that fits the real job. Small does not mean familiar.

Use a cookie when the browser must send a small value with matching HTTP requests. Use `localStorage` for a small string preference. Use `sessionStorage` for a small tab-scoped value. Use

IndexedDB for structured records, Cache Storage for request and response pairs, and OPFS for origin-private files.

Do not put a large client-only object in a cookie. It adds bytes to matching requests. Do not put hundreds of notes in `localStorage`. Its synchronous calls block JavaScript, and every value is a string.

The field-notes app will use several APIs on purpose. Combining tools is normal when each stored value has a clear owner and cleanup rule.

Choose an API for a theme, authentication session, one-tab draft, searchable offline notes, cached manual, and large attachment. Reject one tempting but wrong alternative for every choice.

### Inspect before you change

Use browser DevTools and a clean test profile to identify storage owners, stale versions, and unexpected data.

Storage bugs are easier when you can see the state. Open DevTools before adding more code.

Use the browser Application or Storage panel to inspect cookies, Local Storage, Session Storage, IndexedDB, Cache Storage, and service workers. The Network panel shows `Set-Cookie` responses and later `Cookie` requests.

Test with a clean origin too. Your normal profile may contain an old schema, a stale cache, or a cookie created by code that no longer exists. A private window helps isolate state, but it is not a perfect substitute for testing every supported browser.

Never share a storage screenshot or HAR file without reviewing it. Cookies, tokens, personal notes, URLs, and response bodies may be visible.

Inspect one real site you control. List every stored item, its apparent owner, and whether it survives reload, a new tab, browser restart, and clearing site data. Redact secrets from your evidence.

### Check your understanding: choose and observe storage

Check storage ownership, browser boundaries, API choices, lifetimes, and inspection before writing application data.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Cookies over HTTP

Trace cookie transport, scope values narrowly, secure sessions, and handle partitioned state.

### Follow the cookie round trip

Trace Set-Cookie from a response into the browser cookie jar and back through matching Cookie request headers.

A cookie connects browser storage to HTTP. That is its defining feature.

A server creates or updates a cookie with the `Set-Cookie` response header. The browser stores it,

checks its scope on later requests, and adds matching values to the `Cookie` request header. Frontend code does not set that request header manually.

```
HTTP/1.1 200 OK
Set-Cookie: theme=dark; Path=/; Max-Age=2592000; Secure; SameSite=Lax

GET /notes HTTP/1.1
Cookie: theme=dark
```

Cookies are intentionally small and travel repeatedly. Store a compact identifier or preference, not a profile, document, or cached response.

The browser does not expose `Set-Cookie` to frontend JavaScript. Inspect it in the Network panel, then inspect the stored cookie and the next matching request.

For a script-visible preference, `document.cookie` is synchronous and awkward. The newer promise-based Cookie Store API is cleaner where your supported browsers provide it. Keep authentication cookies `HttpOnly` either way.

Set one harmless preference cookie from a local server. Capture the response and the next request, then change the request path until the cookie no longer matches.

### Control cookie scope and lifetime

Use host-only defaults, narrow paths, Max-Age, and matching deletion attributes instead of creating accidental cookies.

A cookie is identified by more than its name. Scope decides which cookie you update or delete.

Omit `Domain` to create a host-only cookie. Add a narrow `Path` when only one part of the site needs it. Use `Max-Age` for a controlled lifetime. A cookie without `Max-Age` or `Expires` is a session cookie, but browser session restore can keep it across a restart.

```
Set-Cookie: editor=compact; Path=/notes; Max-Age=3600; Secure; SameSite=Lax
Set-Cookie: editor=; Path=/notes; Max-Age=0; Secure; SameSite=Lax
```

To delete the intended cookie, match the original name, domain, and path. A different path creates or removes a different cookie with the same visible name.

`Path` controls delivery. It is not a security boundary. Code from another path on the same origin may still affect JavaScript-readable cookies.

Create two cookies with the same name and different paths. Observe which requests receive each one, then delete both using their exact scopes.

### Lock down a session cookie

Create a host-bound session cookie that resists script reads, insecure transport, and unnecessary cross-site delivery.

A session cookie carries authority. Give it the narrowest useful access.

Set authentication cookies on the server. Use `Secure`, `HttpOnly`, an appropriate `SameSite` value, a controlled lifetime, and no broad `Domain`. The `__Host-` prefix adds browser-enforced rules: `Secure`, `Path=/`, and no `Domain`.

```
Set-Cookie: __Host-session=opaque-random-id; Path=/; Max-Age=1800; Secure; HttpOnly; SameSite
```

`HttpOnly` prevents `document.cookie` from reading the value. It does not stop injected JavaScript from performing authenticated actions. Prevent XSS and protect state-changing cookie requests from CSRF too.

Keep account data and authorization on the server. The cookie should contain an opaque identifier, not a trusted role, email address, or password.

Rotate the identifier after login and privilege changes. Revoke its server record on logout instead of waiting for browser expiration.

Inspect a session cookie from an application you control. Justify its name prefix, transport, script access, same-site behavior, scope, lifetime, rotation, and server-side revocation.

### Handle cross-site and partitioned cookies

Distinguish SameSite rules from partitioned cookies and avoid depending on unrestricted third-party state.

Embedded content creates a second context: who set the cookie, and which top-level site contains it.

`SameSite=Lax` or `Strict` limits cross-site delivery. `SameSite=None` permits cross-site use and requires `Secure`. A `Partitioned` cookie adds the top-level site to its storage key, so the same embedded service receives a separate cookie jar on different sites.

```
Set-Cookie: __Host-widget=compact; Path=/; Secure; SameSite=None; Partitioned
```

Partitioning supports an embedded feature that needs per-site state without one identifier following the user across every host site. It does not turn a cookie into shared first-party storage.

Browser privacy policies and storage access rules keep changing. Design embeds to work without cross-site cookies where possible, and test the supported browser matrix with blocking enabled.

Draw the cookie keys for one widget embedded on two unrelated top-level sites. Explain what the widget server receives with an unpartitioned cookie, a partitioned cookie, and no cross-site cookie access.

### Check your understanding: cookies over http

Check cookie transport, scope, lifetime, security attributes, same-site behavior, and partitioned third-party state.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Web Storage

Use localStorage and sessionStorage for small values, then handle invalid data and tab conflicts.

### Save a small preference

Store and restore one replaceable string preference with localStorage without turning it into an application database.

`localStorage` fits a small preference that should survive a browser restart. Our theme is a good example.

The API is synchronous and stores strings. Write the preference when it changes, then read it during startup. A missing key returns `null`, so define a safe default.

```
localStorage.setItem('field-notes:theme', 'dark')

const theme = localStorage.getItem('field-notes:theme') ?? 'light'
document.documentElement.dataset.theme = theme
```

Prefix keys with the application and feature. This prevents a generic key such as `theme` from colliding with another script on the same origin.

Keep this startup read small. Web Storage blocks JavaScript while it reads or writes. IndexedDB is the better choice for a growing collection.

Only accept known theme values before applying them. A persisted string is input, not trusted configuration.

Save a theme, reload, and prove it returns. Then delete the key and prove the default applies without throwing or rendering an invalid value.

### Keep a draft in one tab

Use sessionStorage for an unfinished editor draft that belongs to one tab and disappears with its page session.

A draft can survive reload without becoming permanent application data. `sessionStorage` gives us that middle ground.

`sessionStorage` uses the same string API as `localStorage`, but it is scoped by origin and top-level tab. A reload keeps the page session. Closing the tab normally ends it.

```
const draftKey = 'field-notes:draft'
editor.value = sessionStorage.getItem(draftKey) ?? ''

editor.addEventListener('input', () => {
  sessionStorage.setItem(draftKey, editor.value)
})
```

Opening another tab creates a separate page session. Duplicating a tab may begin with a copy, but later changes do not create one shared draft. Test the browsers you support.

Remove the draft after a successful save. Otherwise an old recovery value can overwrite or confuse newer saved content.

Type a draft, reload, open a second tab, and close the first tab. Record exactly where the draft appears, then clear it after simulating a successful note save.

### Version and validate stored JSON

Serialize a small object deliberately, reject malformed or outdated data, and migrate its shape instead of trusting persisted strings.

Web Storage stores strings. JSON gives a string structure, not a guarantee that old or edited data

is valid.

Add a version to the stored value. Parse inside `try...catch`, check the expected fields, and fall back safely. When the shape changes, migrate known older versions or discard replaceable data.

```
const raw = localStorage.getItem('field-notes:settings')
let settings = { version: 1, theme: 'light' }

try {
  const saved = JSON.parse(raw ?? 'null')
  if (saved?.version === 1 && ['light', 'dark'].includes(saved.theme)) {
    settings = saved
  }
} catch {}
```

Any script on the origin, a browser extension, or the user through DevTools can change this value. Treat it as untrusted input even though your application wrote it first.

Do not store authentication tokens here. An injected script running in the origin can read and exfiltrate JavaScript-accessible storage.

Save valid settings, malformed JSON, an unknown version, and a theme outside the allowlist. Prove startup uses a safe value in every case and never leaves the interface half-initialized.

### Coordinate tabs and handle failures

React to storage events, recognize last-write-wins conflicts, and keep the interface usable when a Web Storage write fails.

Two tabs can edit the same `localStorage` key. The API offers notification, not conflict resolution.

The `storage` event fires in other documents sharing the storage area. It does not fire in the tab that made the change. Use it to refresh replaceable preferences or warn about stale state.

```
window.addEventListener('storage', event => {
  if (event.key === 'field-notes:theme' && event.newValue) {
    document.documentElement.dataset.theme = event.newValue
  }
})
```

A read followed by a write is not a transaction. Two tabs can both read an old value and overwrite each other. Keep shared updates simple or move structured concurrent state to IndexedDB or a server.

Storage access and writes can throw because of policy, quota, or environment constraints. Catch failures at the write boundary and let the user continue without pretending the value was saved.

Open two tabs and change the theme in each. Confirm which tab receives each event. Then force a failed write and make the interface show a clear non-persistent state.

### Check your understanding: web storage

Check localStorage, sessionStorage, JSON boundaries, failures, and cross-tab synchronization.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## IndexedDB

Store structured notes with keys, indexes, transactions, and versioned schema upgrades.

### Open a notes database

Create the field-notes IndexedDB database and its first object store with a small promise wrapper.

Saved notes are structured records that can grow. IndexedDB is the right browser database for this job.

The native API is event-based. We will use the small `idb` wrapper to work with promises while keeping IndexedDB concepts visible. Install it, open version 1, and create the store during the upgrade transaction.

```
import { openDB } from 'idb'

export const db = await openDB('field-notes', 1, {
  upgrade(db) {
    db.createObjectStore('notes', { keyPath: 'id' })
  },
})
```

An object store is a collection of records. The `id` property is our primary key. Schema changes such as creating stores and indexes belong in a version upgrade, not ordinary page code.

Open the database only in browser code. Server rendering environments do not provide IndexedDB.

Install `idb`, open version 1, and inspect the database in DevTools. Reload twice and prove the upgrade callback does not recreate the existing store.

### Write and read structured notes

Store real JavaScript values with put(), retrieve them by key, and distinguish insert-only behavior from replacement.

IndexedDB uses the structured clone algorithm. We can store a note with a `Date` without converting the whole record to JSON.

Use `add()` when an existing key should be an error. Use `put()` when the operation may insert or replace. Read one record with `get()`, and handle `undefined` when it does not exist.

```
const note = {
  id: crypto.randomUUID(),
  title: 'Trail conditions',
  body: 'The north path is wet',
  updatedAt: new Date(),
}

await db.put('notes', note)
const saved = await db.get('notes', note.id)
```

Functions and DOM elements cannot be cloned into the database. Store data, not live interface objects or behavior.

Do not call `getAll()` forever as the collection grows. A useful app needs indexes, bounds, and pagination or cursors.

Keep record IDs stable across edits. Replacing an ID creates a second note instead of updating the original record.

Add three notes, replace one with `put()`, then try the same key with `add()`. Record the returned record and the constraint failure.

### Query notes through an index

Add a title index in a versioned upgrade and query a bounded subset instead of loading the entire store.

Primary keys answer one access pattern. An index supports a different lookup without scanning every record in application code.

Open database version 2 and add a normalized `titleKey` index. Existing records need a data migration or a fallback path because an index cannot invent missing property values.

```
export const db = await openDB('field-notes', 2, {
  upgrade(db, oldVersion, newVersion, tx) {
    if (oldVersion < 1) {
      db.createObjectStore('notes', { keyPath: 'id' })
    }
    if (oldVersion < 2) {
      tx.objectStore('notes').createIndex('by-title', 'titleKey')
    }
  },
})
```

```
const matches = await db.getAllFromIndex('notes', 'by-title', 'trail', 20)
```

Version checks let a user move directly from any older version. Never assume every browser opened the immediately previous release.

Indexes speed a known read and add work to writes. Add one because the interface queries it, not because the property exists.

The fourth argument bounds this exact-key lookup to twenty records. A growing search interface still needs a deliberate cursor or pagination design.

Upgrade to version 2, migrate existing notes with lowercase `titleKey` values, and query the index. Test both a fresh database and a direct upgrade from version 1.

### Use transactions and unblock upgrades

Group related writes atomically and close stale connections when another tab needs a newer database version.

IndexedDB writes already run in transactions. Use one explicit transaction when several records form one application operation.

Open a `readwrite` transaction, perform the related requests, then await `tx.done`. If one request aborts the transaction, none of its writes commit.

```
const tx = db.transaction(['notes', 'changes'], 'readwrite')
await tx.objectStore('notes').put(note)
await tx.objectStore('changes').add({ noteId: note.id, type: 'saved' })
await tx.done
```

Do not wait for unrelated network work in the middle of a transaction. IndexedDB transactions can become inactive when no database request remains pending.

Another open tab can block a version upgrade. Use the wrapper's `blocking()` callback to close the old connection and tell the user to refresh rather than leaving the upgrade hanging.

Create a second `changes` store in version 3. Force its write to fail and prove the note does not commit. Open an older tab, trigger version 3 elsewhere, and handle the blocked upgrade clearly.

### Check your understanding: indexeddb

Check IndexedDB databases, object stores, keys, indexes, transactions, and versioned upgrades.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

## Offline files and durability

Cache responses, measure quota, request persistence, store files, and design recovery.

### Cache request and response pairs

Store a fetched help page in Cache Storage, return a cloned response, and version the cache deliberately.

Cache Storage is not a general object database. It maps requests to responses.

Open a named cache, fetch a resource, verify the HTTP response, and store a clone. Response bodies are streams, so cloning lets the application keep one readable response while the cache consumes the other.

```
const cache = await caches.open('field-notes-help-v1')
const response = await fetch('/field-guide.json')

if (!response.ok) throw new Error(`HTTP ${response.status}`)
await cache.put('/field-guide.json', response.clone())
const guide = await response.json()
```

A `404` response does not make `fetch()` reject. Check `response.ok` before caching a result you consider successful.

Cache names are your migration boundary. When formats or assets change, create a new cache and remove old names after the replacement is ready.

Cache Storage does not apply HTTP freshness rules for you. Your application or service worker decides when a stored response is stale.

Cache the field guide, disconnect the network, and read it with `cache.match()`. Then simulate a 404 and prove the failed response does not replace good cached data.

### Measure quota and request persistence

Estimate origin usage, handle quota failures, and request persistent storage only when the product can justify it.

Browser storage is best-effort by default. "Persistent" never means immortal.

Use `navigator.storage.estimate()` to display approximate usage and quota. The browser may pad or round these values for privacy. Ask for persistence when data is important, user-created, and not safely stored elsewhere.

```
const { usage = 0, quota = 0 } = await navigator.storage.estimate()
const alreadyPersistent = await navigator.storage.persisted()
const persistent = alreadyPersistent || await navigator.storage.persist()

console.log({ usage, quota, persistent })
```

`navigator.storage.persist()` returns whether persistence was granted. Browsers use their own permission and engagement rules. Keep the app useful when the answer is false.

Writes to IndexedDB, Cache Storage, or OPFS can fail with `QuotaExceededError`. Delete replaceable caches first, preserve user work, and explain the recovery choice.

Do not turn an approximate quota into a promise. Available space and browser policy can change after you display it.

Add a storage status panel with estimated usage, quota, and persistence. Simulate a quota failure and make cleanup remove cached help before any user note or attachment.

### Store an attachment in OPFS

Write and read an origin-private attachment file without confusing browser-managed files with user-visible documents.

OPFS gives an origin a private filesystem. Its files are not normal files the user browses in Finder or Explorer.

Get the origin root, create a file handle, open a writable stream, write the contents, and close the stream. The data shares the origin's storage quota and disappears when the user clears site data.

```
const root = await navigator.storage.getDirectory()
const file = await root.getFileHandle('trail.txt', { create: true })
const writable = await file.createWritable()
await writable.write('North trail closed after rain')
await writable.close()

const saved = await file.getFile()
console.log(await saved.text())
```

Use IndexedDB for searchable note metadata and OPFS for file bytes. Store a stable attachment identifier in the note instead of pretending a file handle is permanent business data.

OPFS avoids a user permission prompt because the files stay inside the origin. Use a download or

file-picker flow when the user expects a visible file they control.

Write one text attachment, link its filename from an IndexedDB note, reload, and read it back. Then delete site data and confirm both the database and file disappear.

### Finish the storage recovery plan

Add explicit export, reset, cache replacement, failure messages, and a tested recovery path to the field-notes app.

The project is not finished when writes succeed. It is finished when loss and change are understandable.

Export user notes and attachments into a portable format. Reset each storage owner deliberately: cookies through their matching server response, Web Storage by known keys, IndexedDB after closing connections, named caches by version, and OPFS entries by identifier.

Do not use one blind "clear everything" call during routine cleanup. Cached responses are replaceable; a field note may not be. Put those values in different cleanup policies and ask before destructive deletion.

Run the final matrix: first visit, reload, second tab, browser restart, offline start, schema upgrade, full quota, rejected persistence, private session, stale cache, export, reset, and restore. Save the evidence for every boundary.

Complete the field-notes app with export and reset controls. Delete all local state, restore from the export, and prove saved notes survive while cached help can be downloaded again independently.

### Check your understanding: offline files and durability

Check Cache Storage, quota estimates, persistence, eviction, OPFS, cleanup, export, and offline recovery.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

---

The interactive version of this course is available at https://flaviocopes.com/courses/browser-storage/.