Where Does a Multi■Page Form “Live” Before You Submit It?

If you’ve ever filled a long form — job application, insurance claim, visa request, checkout flow — you’ve probably felt this fear:

“If my browser crashes right now… I’m going to become a minimalist and move to the mountains.”

Multi■page forms (often called **wizard forms**) need a place to store your answers **before** they become “official” in the database.

In product/engineering language, that place is usually called **draft storage**.

This article explains draft storage in a way that works for:

- non■technical readers (plain language + analogies)
- technical readers (correct terms, common architectures, tradeoffs)

Table of Contents

The core idea: draft vs final (email draft analogy)

Think of a multi■page form like writing an email:

- You type and edit → **draft**
- You click “Send” → **final**

The database record is the “sent email.”

Everything before submit is **draft state**. It is real data, but it is not the final record yet.

Key terms you’ll hear (and what they mean)

- **Wizard form / multi-step form**: form split into steps/pages.
- **Draft**: a partially completed submission.
- **Autosave**: saving the draft periodically or on step change.
- **Session**: a temporary “conversation” between browser and server (often via cookie).
- **Session store**: where server sessions live (often Redis).
- **TTL (Time To Live)**: automatic expiration time for draft/session data.
- **Idempotency**: retrying a request won’t create duplicates.
- **Optimistic concurrency**: detect “someone else edited this draft” using a version number.
- **PII**: personally identifiable information.

If you can use these words comfortably, you sound like you’ve done this in production (even if you haven’t).

What problems draft storage must solve

A good draft system handles:

- **Next/Back navigation** without losing fields
- **Refresh / crash / reconnect** without losing progress
- **Validation** per step and on final submit
- **Security & privacy** for sensitive fields
- **Resuming later** (sometimes across devices)
- **Analytics** (where do users drop off?)

Where can the draft live? (the 4 main architectures)

Option 1 — Browser memory (in-memory state)

Where it lives: your page's runtime state (React state, Vue state, etc.)

Pros

- fastest and simplest

Cons

- refresh/close tab → draft is gone

Best for

- tiny forms
- low stakes data

Real-world vibe: "If you blink, it forgets."

Option 2 — Browser persistence (localStorage / sessionStorage / IndexedDB)

Where it lives: saved on the user's device.

Common options:

- **sessionStorage**: survives refresh, usually dies when tab/browser closes

- **localStorage**: survives browser restart
- **IndexedDB**: better for larger drafts, offline support

Pros

- refresh-proof, sometimes offline-friendly
- no backend needed for drafts

Cons

- **security**: if you store sensitive info and you get XSS, it can leak
- device-bound: switching devices won't bring the draft
- users can clear it

Best for

- "save locally" experiences
- low-to-medium sensitivity data

Analogy: you wrote your draft on a sticky note stuck to *this* laptop.

Option 3 — Server session storage (draft in a session store like Redis)

Where it lives: backend, keyed by a session id (often in a cookie).

Typical stack:

- browser has a session cookie
- server stores draft under that session id in **Redis** (or similar)

Pros

- draft doesn't sit in the browser's localStorage
- central control: TTL, encryption at rest, auditing

Cons

- session lifecycle complexity

- scaling concerns (session store capacity)
- cross-device resume is tricky unless you tie drafts to a user account

Best for

- authenticated apps
- sensitive data
- when you need strong control and short-lived drafts

Analogy: you're writing your draft on the company's whiteboard (it's safer, but the office locks at night).

Option 4 — Drafts table (or "staging" records) in the database

Where it lives: a dedicated drafts table (not the final table).

Typical schema:

- `form_drafts(draft_id, user_id, status, current_step, payload_json, created_at, updated_at, expires_at)`

Pros

- very reliable
- easy "resume later" across devices
- supports auditing and compliance

Cons

- you must design cleanup (TTL/expiration)
- partial/incomplete data is normal — don't let downstream treat it as final

Best for

- long, high-stakes forms (gov, insurance, finance)
- regulated workflows
- anything with "Save draft and continue later" as a feature

Analogy: your draft lives in a “draft folder” on the server, not in the “sent mail” folder.

The most common production pattern: server draft + autosave

If I had to bet on what you’ll see in most serious apps:

1) User starts form → server creates a draft and returns `draft_id` 2) Each step (or every N seconds) → client sends partial updates 3) Final submit → server validates everything and writes the final records

The API usually looks like this

- `POST /drafts` → create draft, returns `draft_id`
- `PATCH /drafts/{draft_id}` → update part of the draft (autosave)
- `GET /drafts/{draft_id}` → resume
- `POST /drafts/{draft_id}/submit` → final validation + write to final tables

Why PATCH? Because you often update “just this step,” not the whole world.

The technical words that matter (and why they matter)

Idempotency (so retries don’t duplicate)

Networks fail. Mobile networks fail more.

If the client sends “save step 2” twice, you still want **one** draft update.

Common technique:

- include an **Idempotency-Key** header
- or use a `draft_version` / `step_version` field

Optimistic concurrency (avoid overwriting)

If the user opens the form in two tabs, you don’t want tab A to overwrite tab B silently.

Common technique:

- store a `version` number in the draft

- require the client to send it back; reject if it’s stale

TTL + cleanup (your drafts will pile up)

Drafts are like coffee cups in an office kitchen. If you don’t clean them, you’ll eventually need a hazmat suit.

Common techniques:

- `expires_at` column + nightly cleanup job
- Redis TTL for session-based drafts

Security (in plain language)

Why localStorage is risky for sensitive data

If an attacker can run JavaScript on your page (XSS), they can read localStorage.

So don’t put:

- passwords
- tokens
- full credit card details
- sensitive PII

If you must store locally:

- minimize what you store
- consider encrypting client-side (but key management becomes its own project)

Server-side draft storage helps

You can:

- encrypt at rest (KMS-managed keys)
- audit access
- enforce ACLs

- expire drafts reliably

Real-world examples

Example 1: Government/insurance form

Requirements:

- save for weeks
- resume across devices
- strong auditing

Typical choice:

- **drafts table in DB** + autosave

Example 2: E-commerce checkout

Requirements:

- fast
- short lived
- handle retries

Typical choice:

- **server session** (Redis) for cart + shipping steps

Example 3: Job application

Requirements:

- save draft for days
- attachments (CV)

Typical choice:

- **drafts table** + object storage for files

How to choose (simple decision chart)

Ask these questions:

1) **Do users need resume after refresh?**

- No → in-memory is OK
- Yes → localStorage/IndexedDB or server drafts

2) **Do users need resume across devices?**

- Yes → server drafts tied to user

3) **Is the data sensitive?**

- Yes → avoid localStorage; prefer server drafts/session store

4) **How long should drafts live?**

- Minutes/hours → session store with TTL
- Days/weeks → drafts table with `expires_at`

Data engineering angle (why DEs should care)

Draft storage affects:

- **data correctness** (no partial records in final tables)
- **lineage** (what was saved vs submitted)
- **funnel analytics** (drop-off by step)
- **compliance** (retention + deletion)

In other words: draft storage is not just a frontend thing.

Conclusion

Multi-page forms need a place to put “unfinished work.” That place is draft storage.

If you remember one line:

> Keep drafts separate from final records, and treat the draft system like a real data product.

In a future article, I’ll go deeper into:

- designing a drafts schema (JSON vs normalized tables)
- autosave performance patterns
- handling file uploads inside multi-step flows
- and the security checklist that prevents very expensive mistakes

If you tell me your use case (public form? logged-in app? sensitive data?), I can recommend the best architecture for your situation. *** End Patch"}Commentary to=functions.apply_patch ■■ json