Lab 07: *Mock DB in Browser* | Module-02

Mock Database in Browser

Local Storage (via Browser API) + Cloud Storage (via JSONBin API)

REQUIRES: HTTP SERVER

Table of Content: *Mock Database in Browser*

Lab Introduction

Goal:

The primary goal of this lab is to build a client-side, mock document database that runs entirely in the browser. This database mimics the API style of MongoDB and is designed to be storage-agnostic for all data operations (Create, Read, Update, Delete), meaning the application code that uses the database doesn't need to know *how* or *where* the data is stored. Important to note, because this database is strictly client-side, **all data should be treated as public.**

Key Learning Objectives:

- **Persist data locally** using the browser's **LocalStorage API**, allowing an application's state to survive page reloads.
- **Persist data to the cloud** using the **JSONBin API**, enabling shared state between different users or devices.
- **API Design:** Creating a clean, reusable Data Layer with a consistent API for Create, Read, Update, and Delete (CRUD) operations.
- **Decoupled Architecture:** Using a swappable "adapter" pattern to separate the core database logic from the underlying storage mechanism (In-Memory, LocalStorage, Cloud Storage).
- **Asynchronous Operations:** Handling data fetching and saving with asynchronous JavaScript, particularly when interacting with cloud services.
- **Database Concepts:** Gaining practical experience with document database concepts like collections, documents, schema design, and query models.

Core Architectural Concepts

To achieve this, our mock database is organized into three clear layers:

1. **The AppDoc (Single Source of Truth):** A single JSON object that holds all your application's data in collections (arrays) of documents (objects) (e.g., `users`, `todos`).

2. **The DB Module (`db.js`)** The public API that your UI calls. It handles the "how" of data manipulation but delegates the "where" to an adapter.

3. **Adapters (Storage Engines):** Swappable modules that handle the physical reading and writing of the AppDoc. The lab develops three adapters
    - **`MemoryAdapter`:** A temporary, volatile, in-memory store (data is lost on page reload).
    - **`LocalStorageAdapter`:** A persistent store using the browser's `LocalStorage`. Data survives page reloads.
    - **`JsonBinAdapter`:** A persistent cloud store using the public JSONBin REST APII, allowing data to be shared between different clients.

By the end, you'll be able to swap from `LocalStorage` to `JSONBin` without changing a single line of your application's CRUD logic. This powerful pattern is the key to building flexible and maintainable software.

Required Tools:

- A modern web browser (Google Chrome is recommended).
- A local web server to run the project. A simple tool like `http-server` for Node.js is perfect.
- A free account to JsonBin

Project Architecture:

Start this project by downloading the starter files from GitHub. See the project structure below.

```
browser-db
├── index.html
├── demos
│   ├── demo-runner.js
│   ├── demo.css
│   ├── jsonbin.html
│   ├── local.html
│   ├── memory-adv.html
│   ├── memory.html
│   └── sync.html
├── scripts
│   ├── db.js
│   ├── model.js
│   ├── sync.js
│   └── adapters
│       ├── jsonBinAdapter.js
│       ├── localStorageAdapter.js
│       └── memoryAdapter.js
├── styles
│   └── mvp.css
└── tests
    ├── console.css
    ├── test-adv-memory.html
    ├── test-local.html
    └── test-runner.js
```

*filenames in red are added/coded by you

Download Starter files:

https://github.com/scalemailted/browser-db

Document Database -- Basics

What is a Document Database?

A document database stores data in flexible, self-contained **documents**, which are typically JSON-like objects. These documents live inside **collections** (the equivalent of SQL tables). Unlike the rigid schemas of relational databases, documents in a collection can have varying structures, making them ideal for evolving applications.

Relational (SQL)	**Document (NoSQL)**
Rows in Tables	**Documents** in **Collections**
Fixed Schema	Flexible Schema
`JOIN`s across tables	**Embedding** data or **Referencing** by ID

The MongoDB Connection

The most popular document database is **MongoDB**. The concepts you'll use in this lab -- collections, field queries, and update operators like `$set` -- are directly inspired by its API. This lab gives you a practical head start on the patterns used in modern, full-stack development.

Why This Model Fits Our Lab

This lab is the perfect introduction to document databases because:

1. We model our entire app's state as a **single JSON document**, a simplified version of this paradigm.
2. You will implement a **Mongo-like API** for all CRUD (Create, Read, Update, Delete) operations.
3. You'll learn core persistence patterns like **serialization** (for LocalStorage) and **optimistic concurrency** (for JSONBin) that mirror how real-world databases operate.

Our Mock DB: Design & API Overview

Our browser database follows a few clear design principles and exposes a simple, powerful API.

Core Design Principles

- **Single Source of Truth**: One in-memory JSON object (the `AppDoc`) holds all collections and acts as the canonical state.
- **Safe Reads & Writes**: Reads always return a **deep copy** of the data, so the original cache cannot be accidentally changed. All writes are performed on a copy and then saved through an adapter in a single step (*write-through*).
- **IDs & References**: Documents use unique string `id`s. Relationships between documents (e.g., a `todo` assigned to a `user`) are handled by storing the target document's `id` as a reference.

CRUD: The Four Core Database Operations

All database interactions can be categorized into four fundamental operations known as **CRUD**. Understanding this model helps clarify the role of each function in our API.

CREATE (Add new data)

- Our primary creation function is `insertOne`.
- The advanced `upsertOne` function also performs creation if a record doesn't already exist.

READ (Retrieve existing data)

- Our API provides several ways to read data, from simple lookups like `findOne` and `findMany` to more powerful queries with `findManyBy` and the combined `find` function.

UPDATE (Modify existing data)

- The basic `updateOne` handles simple changes.
- For more complex needs, `updateOneOps` provides helper operators, and `upsertOne` will update a record if it's found.

DELETE (Remove existing data)

- `deleteOne` is our function for removing a document from a collection.

The `transact` function is a higher-level utility that allows you to batch multiple **C**reate, **U**pdate, and **D**elete operations into a single, atomic sav

API Surface (Mini-Mongo)

This is the set of functions you will build in `db.js`. Your application's UI will call these methods to interact with the data, regardless of which storage adapter is active.

```
/* ================================
   Setup & Initialization
   ================================ */
useAdapter(adapter)        // Choose the storage engine (e.g., LocalStorage)
boot()                     // Load the AppDoc from the adapter into memory


/* ================================
   Basic CRUD Operations (Dev Cycle 1)
   ================================ */
insertOne(col, data)       // Create a new document in a collection
findMany(col, predicate?)  // Read multiple documents using a JS predicate function
findOne(col, predicate)    // Read the first matching document using a predicate
updateOne(col, id, patch)  // Update a document with a shallow patch
deleteOne(col, id)         // Delete a document by its ID


/* ================================
   Advanced CRUD Operations (Dev Cycle 2)
   ================================ */
// Advanced Reads
findManyBy(col, filter)    // Read documents using a Mongo-style filter object
findOneBy(col, filter)     // Read one document using a filter object
find(col, args)            // Combined query for filtering, sorting, paging, & projection

// Advanced Writes
updateOneOps(col, id, ops) // Update a doc using operators like $set and $addToSet
upsertOne(col, filter, data) // Create a doc or update it if it exists
transact(fn)               // Perform multiple mutations in a single, atomic save
```

Designing the Data Schema -- Mocked for a TODO App

The `AppDoc`: Our Single Source of Truth

In this lab, the entire application's state is stored in a single JSON object called the `AppDoc`. This object lives in memory during runtime and is what gets saved and loaded by our storage adapters.

The `AppDoc` contains several **Collections**, which are simply arrays of related **Documents** (JavaScript objects).

- Each **Document** has a unique string `id`.
- To create relationships, we **Reference** other documents by storing their `id`s (e.g., a `todo`'s `assignedTo` field holds a user's `id`).
- For tightly coupled data, like a `todo`'s `subtasks`, we **Embed** the data directly within the parent document.

Versioning for Concurrency

To handle updates and prevent data conflicts (especially with the JSONBin adapter), every `AppDoc` includes two special versioning fields at its root:

- `rev` (integer): Increments on every successful save. Used for optimistic concurrency checks.
- `updatedAt` (ISO string): A timestamp of the last modification.

The Mock TODO App Schema

Here is the concrete structure of our application's data.

Collections:

`users=[]  | projects=[]  | todos=[]  | tags=[]  | comments=[]`

Document Shapes:

- **users:** `{ id, name }`
- **projects:** `{ id, name, ownerId, tagIds: string[] }`
- **todos:** `{ id, projectId, title, done, assignedTo, due?, tagIds: string[], subtasks: {id, title, done}[], commentIds: string[] }`
- **tags:** `{ id, name }`
- **comments:** `{ id, todoId, authorId, text, ts }`

The Seed Data (Mocked)

To ensure you have data to work with from the start, the lab begins with the following pre-populated `AppDoc`. This small scenario gives every feature something real to act on.

```
{
    version: 1,
    rev: 1,
    updatedAt: "2025-10-01T15:00:00.000Z",
    users: [
      { id: "u-alice", name: "Alice" },
      { id: "u-bob",   name: "Bob" }
    ],
    projects: [
      { id: "p-1", name: "Course Site", ownerId: "u-alice", tagIds: ["t-web"] }
    ],
    todos: [
      {
        id: "t-1",
        projectId: "p-1",
        title: "Set up repo",
        done: false,
        assignedTo: "u-alice",
        due: "2025-10-10",
        subtasks: [{ id:"s-1", title:"README", done:true }, { id:"s-2", title:"CI", done:false }],
        commentIds: ["c-1"],
        tagIds: ["t-setup"]
      }
    ],
    comments: [
      { id: "c-1", todoId: "t-1", authorId: "u-bob", text: "Add badges", ts: "2025-10-01T15:00:00Z" }
    ],
    tags: [
      { id: "t-web",   name: "web" },
      { id: "t-setup", name: "setup" }
    ]
}
```

The Core Data Flow & Adapter Contract

Data Flow Scheme

Understanding the flow of data is the key to this lab's architecture. Every database operation, from reading to writing, follows a clear and predictable sequence. This ensures that our data remains consistent and that our components are cleanly decoupled.

How a Write Operation Works (e.g., `updateOne`)

Every time you modify data, the system performs the following four steps:

1. **Get a Safe Copy:** The operation begins by taking a **deep copy** of the in-memory `AppDoc`. This prevents any accidental changes to the live application state.
2. **Mutate the Copy:** All changes (e.g., updating a field, adding a record) are applied to this *copy*, not the original.
3. **Persist via Adapter:** The modified copy is handed to the active storage **adapter's** `save()` method. The adapter is responsible for writing the data to its destination (Memory, LocalStorage, or JSONBin) and updating the versioning fields (`rev` and `updatedAt`).
4. **Refresh the Cache:** Once the adapter confirms a successful save, the DB module replaces the old in-memory `AppDoc` with the newly saved version. Future reads will now see the updated state.

This "copy-on-read, write-through" pattern is central to keeping our data safe and predictable.

The Adapter Contract

This entire flow is possible because you write your app code against a single DB API, which in turn relies on a simple, stable **contract** that every storage adapter must follow.

The "magic" here isn't inheritance—it's **composition**. By adhering to this contract, we can swap storage engines (`Memory` ↔ `LocalStorage` ↔ `JSONBin`) without changing any UI or CRUD logic.

Every adapter you build will implement this tiny, powerful contract:

- **`load()`**: Reads the latest `AppDoc` from the storage medium.
- **`save(doc)`**: Writes a given `AppDoc` to the storage medium.
- **`reset()`**: (Optional) Clears the data from the storage medium.
- **`snapshot()`**: (Optional) Returns a read-only copy of the data directly from storage for debugging.

Lab Roadmap & Core Principles

Overview

This lab is broken down into five major development cycles. Before you begin, understand this roadmap, the core principles you must follow, and how your work will be tested.

The Five Development Cycles

1. **Build Basic CRUD (In-Memory):** First, you'll build the basic `Create`, `Read`, `Update`, and `Delete` API and prove the core database contract works using a temporary, in-memory adapter.
2. **Implement Advanced CRUD (In-Memory):** Next, you'll extend the API with powerful, Mongo-like features for advanced queries and updates, still without any persistence.
3. **Add Local Persistence (LocalStorage):** Then, you will add persistence by building a `LocalStorageAdapter`, allowing your app's data to survive page reloads.
4. **Add Cloud Persistence (JSONBin):** After that, you'll connect your database to the cloud by implementing a `JsonBinAdapter` for shared, remote data storage.
5. **Sync Multiple Adapters (Local ↔ Cloud):** Finally, you will demonstrate the power of the adapter pattern by implementing sync functionality between your local and cloud storage.

Core Principles for Success (The Golden Rules)

To succeed in this lab, you must adhere to the following architectural rules in every cycle. These principles ensure your data layer is robust, decoupled, and safe.

- **Separation of Concerns:** Application code must only talk to `db.js`. Never interact directly with `LocalStorage` or `fetch` in your UI.
- **Single Source of Truth:** The in-memory `AppDoc` is the canonical state. All reads and writes must go through the DB API.
- **Safe Reads (Copy-on-Read):** Read operations (`getDoc`, `find`, etc.) must always return a **deep copy** of the data to prevent accidental mutation of the cache.
- **Safe Writes (Write-Through):** Every function that mutates data (`insertOne`, `updateOne`, etc.) must end by calling `adapter.save(doc)` to persist the change.
- **Consistent Versioning:** The storage adapters are responsible for stamping every successful save with an incremented `rev` and an updated `updatedAt` timestamp.

How to Verify Your Work (Pass Criteria)

This project includes two folders to help you test your code at every step:

- **`tests/`:** These are automated approval pages. For each milestone, you should be able to open the corresponding test file (e.g., `test-memory.html`) and see all green checkmarks (✓).
- **`demos/`**: These are interactive consoles. They allow you to run sample commands, edit the inputs, and see the live results of your database API in action.

Dev Cycle 1:

Basic CRUD Operations

(Volatile In-Memory)

What you're building (and why)

You'll implement a **browser-only document database** that acts as the **single source of truth** for your app. In this cycle, the DB is **volatile (in-memory)** -- perfect for fast iteration and proving the **DB contract** (CRUD) before tackling persistence.

Objective for Dev Cycle 1

- **Boot** the DB into memory
- **Create** records (`insertOne`)
- **Read** records (`getDoc`, `findMany`, `findOne`)
- **Update** records (`updateOne`)
- **Delete** records (`deleteOne`)

Verification:

each goal ends with **Approve** steps that map to:

- **Tester:** `tests/test-memory.html`
- **Demo:** `demos/memory.html` (interactive console)

Table of Contents

Goal 0: Stub Browser Database with CRUD methods

'Approach' → Plan phase

Goal:

Stub out the full set of functions that define Basic CRUD for our mock document database so the testers and demos can import the API immediately. Nothing executes yet -- each function exists and throws a clear “not implemented” error -- ensuring the code compiles, the pages load, and we can implement each function in the next goals.

Why this matters

The demos and testers are wired to call these functions; by exporting them now (even empty), we verify the integration surface and avoid “missing export” errors, while making it obvious when a function hasn’t been implemented yet.

'Apply' → Do phase

Step 1: Stub function -- boot

Responsibility: Loads the current application document from the selected adapter into memory and establishes it as the **single source of truth**; returns a **read-only snapshot** (a deep copy) of that document.

`scripts/` → *db.js*

```
/* ========================
   Scaffold & Boot
   ======================== */

// load the app document via the adapter and cache it
export async function boot() {
  throw new Error("boot() not implemented yet"); // placeholder
}
```

Step 2: Stub function -- create operations

Responsibilities: Creates a new **record in the target collection** (adds to the array inside the current AppDoc), assigns a unique `id`, and **saves the updated Doc** through the adapter; returns the inserted record (including its `id`).

`scripts/` → *db.js*

```
/* ========================
   CREATE
   ======================== */

// insert a new record into collection `col`
export async function insertOne(col, data) {
  throw new Error("insertOne() not implemented yet");
}
```

Step 3: Stub function -- read operations

`getDoc()` — Returns a **deep copy snapshot** of the entire document (never a live reference), suitable for read-only use in views and tests.
`findMany(col, predicate?)` — Returns all records in a collection that match an optional **predicate** (or **all** if omitted); never mutates data.
`findOne(col, predicate)` — Returns the **first** record that matches a predicate or **null** if none; never mutates data.

scripts/→ *db.js*

```
/* ========================
   READ
   ======================== */

// get a safe copy of the cached doc
export function getDoc() {
  throw new Error("getDoc() not implemented yet");
}

// read many
export function findMany(col, pred = () => true) {
  throw new Error("findMany() not implemented yet");
}

// read one
export function findOne(col, pred) {
  throw new Error("findOne() not implemented yet");
}
```

Step 4: Stub function -- update operations

Responsibilities: Locates a record by `id` in the target collection and applies a **shallow** patch (top-level fields only; **arrays are replaced**), then performs a **write-through** save; returns `1` when a record was updated or `0` if no match was found.

scripts/ → *db.js*

```
/* ========================
   UPDATE
   ======================== */

// apply shallow patch; arrays are replaced
export async function updateOne(col, id, patch) {
  throw new Error("updateOne() not implemented yet");
}
```

Step 5: Stub function -- delete operations

Responsibilities: Removes the record with the given `id` from the target collection and performs a **write-through** save if anything changed; returns `1` when a record was deleted or `0` if no match was found.

scripts/ → *db.js*

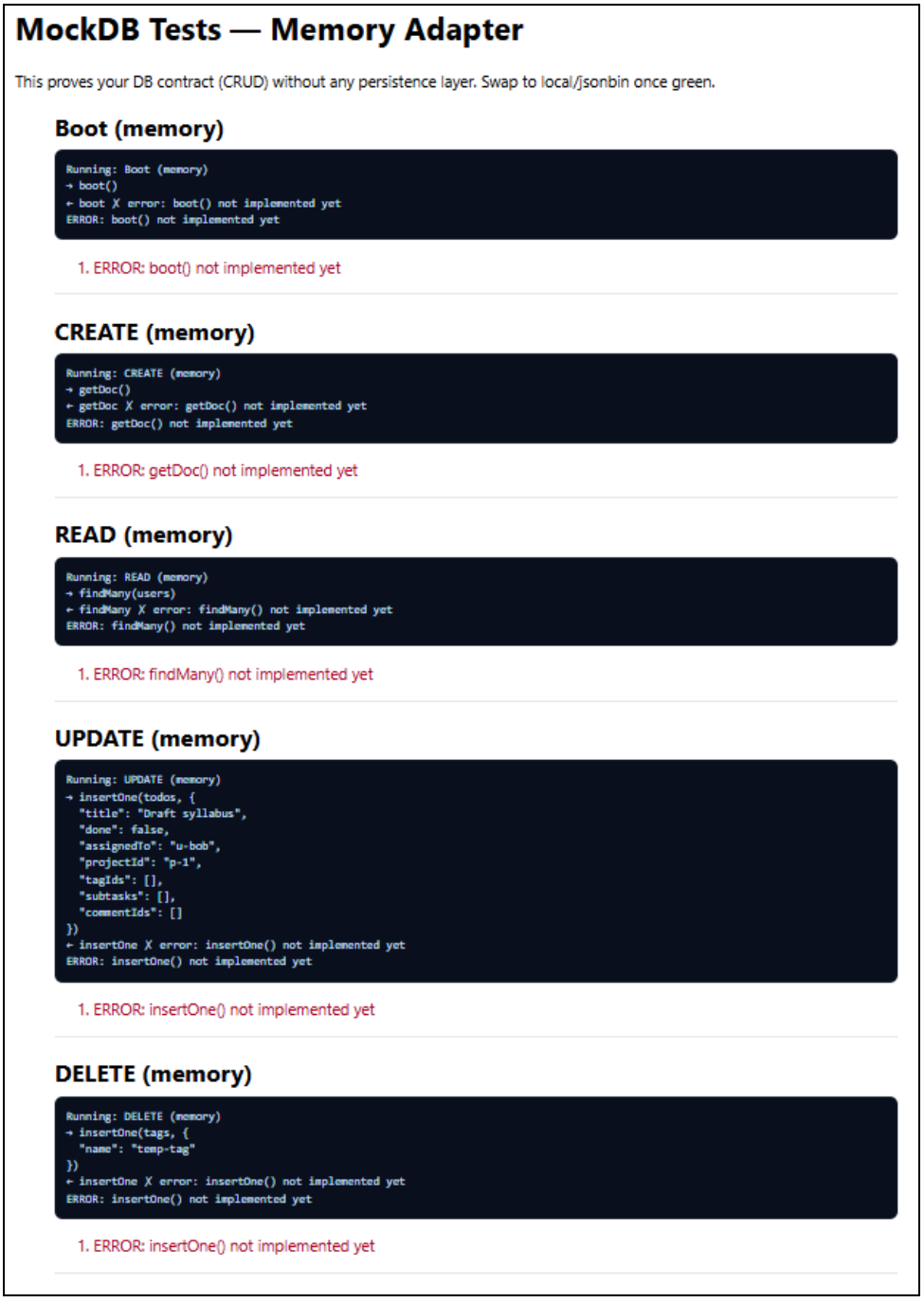
```
/* ========================
   DELETE
   ======================== */

export async function deleteOne(col, id) {
  throw new Error("deleteOne() not implemented yet");
}
```

'Approve' → Test phase

Test Instructions:

1. **Serve over HTTP** (from project root): e.g. `npx http-server`
2. Open `http://localhost:<PORT>/tests/test-memory.html` (or click **Test — Memory Adapter (no persistence)** on the launcher).
3. The page runs **Boot** → **CREATE** → **READ** → **UPDATE** → **DELETE**; green ✓ = pass, red ✗ = fail.

URL → http://localhost:8080/tests/test-memory.html



Goal 1: Browser Database - Boot Operation

'Approach' → Plan phase

What we're achieving

- **Milestone 1 (Adapter readiness):** Build a tiny **MemoryAdapter** that can **load** a seeded app document and **save** changes (stamping `rev/updatedAt`). This proves the *storage contract*.
- **Milestone 2 (DB boot):** Wire the DB layer to **select an adapter** and **boot** the app document into memory as the **single source of truth**.

Design at a glance

- **Adapter seam:** The DB never talks to storage directly; it only calls `adapter.load()` / `adapter.save(next)`.
- **Single source of truth:** On boot, we cache the loaded doc in `_doc`. All reads/writes go through the DB API (later goals).
- **Stamping discipline:** Adapters stamp `rev/updatedAt` on successful saves.
- **Read safety:** When we return the doc to callers, we'll return **copies** (no mutation leaks). For Goal 1, we just load and hold.

'Apply' → Do phase

Milestone 1: MemoryAdapter (establish the storage contract)

Goal:

Objective: Implement an in-memory adapter with the same contract we'll use for persistent adapters later.

Outcome (success means):

- `load()` returns a seeded document the first time and the current doc thereafter.
- `save(next)` updates the in-memory doc and stamps `rev/updatedAt`.
- `reset()` clears the in-memory doc; next `load()` reseeds.
- `snapshot()` returns a deep copy (for debugging/tests).

Step 1: Class shell

Define an **in-memory storage engine** that holds the AppDoc in RAM and exposes the same adapter contract you'll use later (load/save/reset/snapshot). It maintains two fields: `this._doc` (the cached AppDoc, initially `null`) and `this._stampOnSave` (whether `save(next)` should bump `rev` and set `updatedAt`). The constructor accepts `{ stampOnSave=true }`, keeping stamping **on** by default so behavior matches LocalStorage/JSONBin.

`scripts/adapters/memoryAdapter.js` → *MemoryAdapter class*

```
// Minimal in-memory adapter for Part 1 - Establish the DB contract without persistence.

import { seedDoc } from "../model.js";

export class MemoryAdapter {
  constructor({ stampOnSave = true } = {}) {
    this._doc = null;
    this._stampOnSave = stampOnSave;
  }
}

export const memoryAdapter = new MemoryAdapter();
```

Step 2: Instance Method -- _stamp()

Add a private helper that **standardizes versioning** on every successful write. `_stamp(d)` increments `d.rev` (or initializes it to `1 + (rev || 0)`) and sets `d.updatedAt` to the current time in **ISO 8601** (UTC) via `new Date().toISOString()`. The adapter will call this right before persisting updates (when `stampOnSave` is enabled), so all adapters share the same revision/last-modified policy.

`scripts/adapters/memoryAdapter.js` → *MemoryAdapter._stamp()*

```
_stamp(d) {
    d.rev = (d.rev ?? 0) + 1;
    d.updatedAt = new Date().toISOString();
}
```

Step 3: Instance Method -- load()

Implement the adapter's **lazy loader**: on the first call, initialize `this._doc` with `seedDoc()`, and on subsequent calls, return the cached document. This method is **idempotent** (no stamping, no mutation) and simply provides the current AppDoc that the DB layer will cache during `boot()`. It's `async` to match the common adapter contract (so swapping to HTTP-based adapters later requires no signature changes).

`scripts/adapters/memoryAdapter.js` → *MemoryAdapter.load()*

```
async load() {
    if (!this._doc) this._doc = seedDoc();
    return this._doc;
}
```

Step 4: Instance Method -- save()

Implement the adapter's **write-through**: accept a fully-formed next AppDoc, optionally **stamp** it (`rev`/`updatedAt`) when `stampOnSave` is enabled, then replace the cached document. This is an in-memory assignment (no I/O), but it

mirrors the contract used by persistent adapters so your DB layer can call `save(next)` consistently across Memory, LocalStorage, and JSONBin.

`scripts/adapters/memoryAdapter.js` → *MemoryAdapter.save()*

```
async save(next) {
    if (this._stampOnSave) this._stamp(next);
    this._doc = next;
}
```

Step 5: Instance Method -- reset()

Provide a **clean-slate control** for tests and demos: `reset()` clears the cached AppDoc by setting `this._doc = null` -- it does **not** stamp, save, or touch any external storage. After a reset, the next `load()` call will lazily **reseed** from `seedDoc()`, giving you a predictable starting state without reloading the page or swapping adapters.

`scripts/adapters/memoryAdapter.js` → *MemoryAdapter.reset()*

```
reset() {
    this._doc = null;
}
```

Step 6: Instance Method -- snapshot()

Expose a **read-only view** of the cached AppDoc by returning a deep copy (`structuredClone(this._doc)`), so callers can inspect state without risking accidental mutation. If no document has been loaded yet, `snapshot()` returns whatever `_doc` currently holds (typically `null`) and **does not** trigger a load or reseed. This is primarily for tests/demos and debugging -- use `getDoc()` in the DB layer for normal read flows.

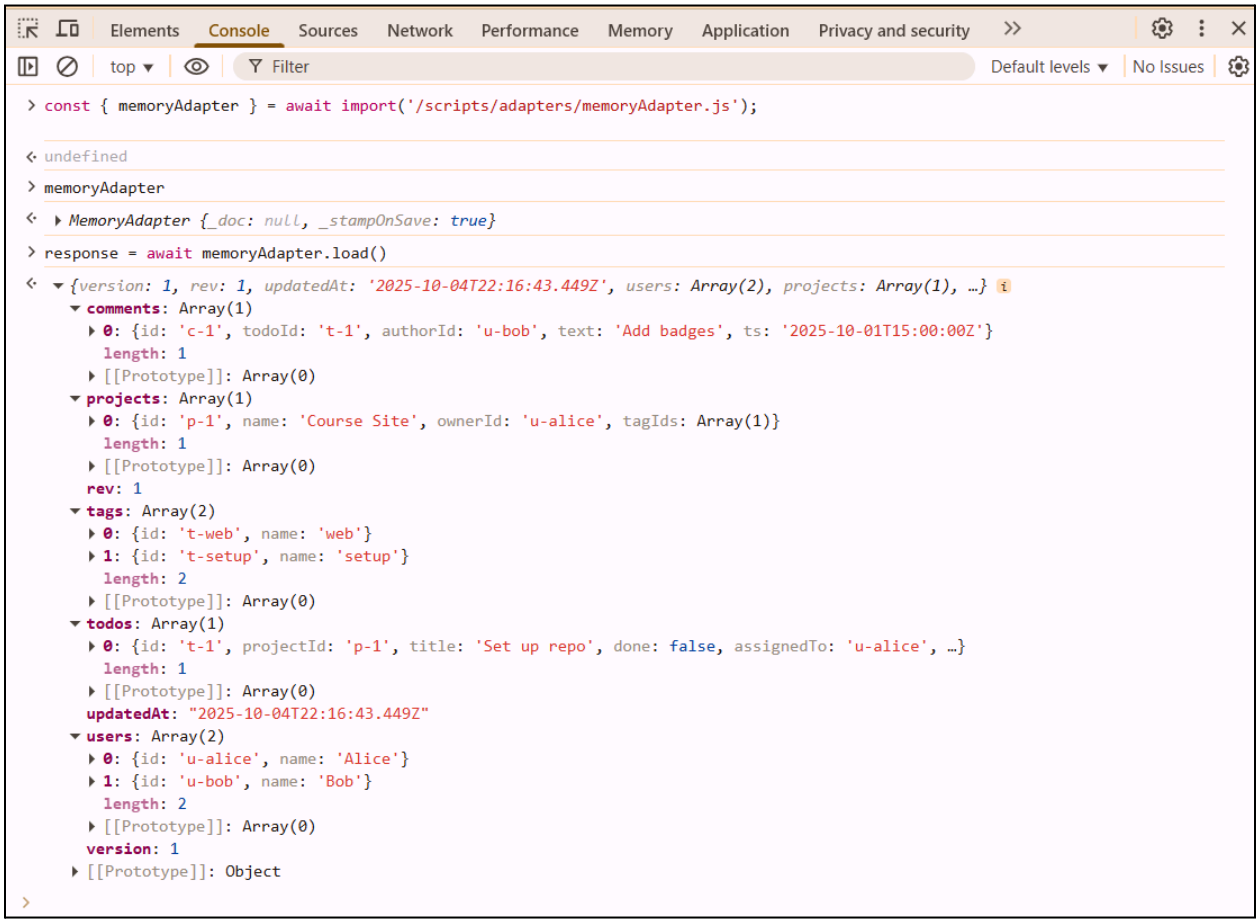
`scripts/adapters/memoryAdapter.js` → *MemoryAdapter.snapshot()*

```
snapshot() {
    return structuredClone(this._doc);
}
```

Milestone 1 -- Testing:

Type these, line by line, in the Console:

```
// 1) Import the adapter (adjust the path if needed)
const { memoryAdapter } = await import('/scripts/adapters/memoryAdapter.js');

// 2) Inspect the instance (you should see MemoryAdapter { _doc: null, _stampOnSave: true })
memoryAdapter;

// 3) Load the document (lazy seed on first call)
const response = await memoryAdapter.load();

// 4) Inspect the loaded doc (expand the object to see users, projects, todos, comments, tags)
response;
```


Elements
Console
Sources
Network
Performance
Memory
Application
Privacy and security
top
Filter
Default levels
No Issues
> const { memoryAdapter } = await import('/scripts/adapters/memoryAdapter.js');
undefined
> memoryAdapter
▸ MemoryAdapter {_doc: null, _stampOnSave: true}
> response = await memoryAdapter.load()
▾ {version: 1, rev: 1, updatedAt: '2025-10-04T22:16:43.449Z', users: Array(2), projects: Array(1), …}
▾ comments: Array(1)
▸ 0: {id: 'c-1', todoId: 't-1', authorId: 'u-bob', text: 'Add badges', ts: '2025-10-01T15:00:00Z'}
length: 1
▸ [[Prototype]]: Array(0)
▾ projects: Array(1)
▸ 0: {id: 'p-1', name: 'Course Site', ownerId: 'u-alice', tagIds: Array(1)}
length: 1
▸ [[Prototype]]: Array(0)
rev: 1
▾ tags: Array(2)
▸ 0: {id: 't-web', name: 'web'}
▸ 1: {id: 't-setup', name: 'setup'}
length: 2
▸ [[Prototype]]: Array(0)
▾ todos: Array(1)
▸ 0: {id: 't-1', projectId: 'p-1', title: 'Set up repo', done: false, assignedTo: 'u-alice', …}
length: 1
▸ [[Prototype]]: Array(0)
updatedAt: "2025-10-04T22:16:43.449Z"
▾ users: Array(2)
▸ 0: {id: 'u-alice', name: 'Alice'}
▸ 1: {id: 'u-bob', name: 'Bob'}
length: 2
▸ [[Prototype]]: Array(0)
version: 1
▸ [[Prototype]]: Object

Milestone 2: Boot the Database with the MemoryAdapter

Goal:

Wire the DB layer to the storage engine and **establish the single source of truth** in memory. Calling `useAdapter(memoryAdapter)` selects the engine; calling `boot()` **loads the seeded AppDoc** via the adapter and caches it in `_doc` so all later CRUD operations read/write the same document. For now `boot()` may return the loaded doc directly; in later goals, reads will return **safe copies** to prevent mutation leaks.

Step 1: DB Module state (private cache & seam)

Declare the private fields that the DB layer will use to talk to storage and hold the current AppDoc.

scripts/ → *db.js*

```
let _adapter = null;
let _doc = null;
```

Step 2: useAdapter(adapter) — choose storage engine

Select the storage implementation for this DB session (Memory now; LocalStorage/JSONBin later).

scripts/ → *db.js*

```
// choose storage engine
export function useAdapter(adapter) {
    _adapter = adapter;
}
```

Step 3: boot() — load & cache the AppDoc

Guard if no adapter, load the document via `adapter.load()`, cache it in `_doc`, and return the loaded doc.

scripts/ → *db.js*

```
// load the app document via the adapter and cache it
export async function boot() {
    if (!_adapter) throw new Error("No adapter set. Call useAdapter(...) first.");
    _doc = await _adapter.load();
    return _doc;
}
```

Milestone 2 -- Testing:

1. **Serve over HTTP** (from project root): e.g. `npx http-server`
2. Open **`http://localhost:<PORT>/tests/test-memory.html`** (or click **Test — Memory Adapter (no persistence)** on the launcher).
3. The page runs **Boot** → **CREATE** → **READ** → **UPDATE** → **DELETE**; green ✓ = pass, red ✗ = fail.

'Approve' → Test phase

Test Instructions:

1. **Serve over HTTP** (e.g., `npx http-server`) and open **`/tests/test-memory.html`** to verify **Boot** passes.

2. Then open the **interactive console** at **`/demos/memory.html`** and try the same calls to see live results.

3. Use **Appendix A — Client-Only DB Cheat Sheet** to paste different query patterns into the demo and observe behavior.

4. **Passing** for this goal: Boot succeeds in the tester.

Goal 2: Browser Database -- Create Operations

'Approach' → Plan phase

Goal

Enable **Create** on our mock document database so apps can add new records safely and predictably. A create must: (1) generate a **unique id**, (2) work against a **safe working copy** (never mutate the cache directly), and (3) **write through** the adapter so `rev/updatedAt` are stamped and the cache stays in sync.

Design for this iteration

- **ID policy:** `uid()` returns a short unique string (e.g., `crypto.randomUUID().slice(0,8)`), so every record in any collection has a stable `id`.
- **Read safety:** `getDoc()` always returns a **deep copy** of `_doc`—callers can't accidentally mutate the cache.
- **Write-through update:** `insertOne(col, data)` builds `{ id: uid(), ...data }` → pushes into `d[col]` (copy) → `await _adapter.save(d)` → sets `_doc = d` → returns **the inserted record**.
- **Stamping location:** The adapter (Memory/LocalStorage/JSONBin) is responsible for **bumping** `rev` and **setting** `updatedAt` on successful saves—consistent across storage engines.
- **Shape assumptions:** We assume `AppDoc[col]` is an **array** (users/projects/todos/comments/tags). If a collection name is invalid, we'll add guards later (Goal 6 "Guardrails," optional).

'Apply' → Do phase

- Milestone 1: Unique ID for New Entries
- Milestone 2: Get the Document (safe copy)
- Milestone 3: Create Operation → insertOne

Milestone 1: Unique ID for New Entries

Step 1: uid() utility

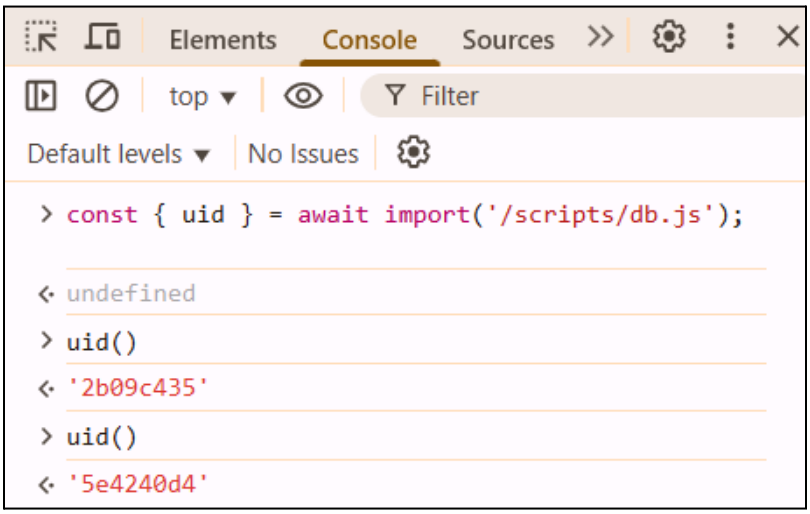
Create a small, portable ID generator so every new record has a unique ID.

`scripts/` → *db.js*

```
export const uid = () => crypto.randomUUID().slice(0, 8);
```

Milestone 1 -- Testing

- In DevTools: `import uid` from `db.js` & try invoking it
- Ensure the page is served over HTTP so `crypto.randomUUID` exists.



Milestone 2: Get the Document (safe copy)

Step 1: getDoc() method

Return a **deep copy** of the cached AppDoc so reads never expose a mutable reference.

`scripts/` → *db.js*

```
// get a safe copy of the cached doc
export function getDoc() {
  // Return a safe copy so callers can't mutate the cached doc directly
  return structuredClone(_doc);
}
```

Milestone 2 -- Testing

- In DevTools: `import getDoc` from `db.js` & try invoking it, then inspecting the returned object

Milestone 3: Create Operation → insertOne

Goal: Add a new record to a collection in the AppDoc

If you come from SQL, this is like:

```
INSERT INTO todos (…) VALUES (…)
```

In our API, you'd write:

```
insertOne("todos",   { title:"Demo", done:false, assignedTo:"u-bob", projectId:"p-1",
                       tagIds:[], subtasks:[], commentIds:[] });
```

Step 1: Build + write-through (safe, ID'd insert)

Create the record with a **unique** `id`, push it into a **safe copy** of the document, then **save** via the adapter (which stamps `rev/updatedAt`) and refresh the cache. Finally, return the **inserted record** so callers can use its `id` immediately.

Notes

- **ID policy:** `uid()` generates the record's primary key; keep it with the record for later reads/updates.
- **Stamping:** Adapters bump `rev` and set `updatedAt` on successful saves--no extra work in `db.js`.
- **Shape assumption:** `AppDoc[col]` is an **array**
- **No mutation leaks:** We modify the **copy** (`d`), never `_doc` directly.

`scripts/` → *db.js*

```
// insert a new record into collection `col`
export async function insertOne(col, data) {
  const d = getDoc();
  const rec = { id: uid(), ...data };
  d[col].push(rec);
  await _adapter.save(d);
  _doc = d;
  return rec;
}
```

Milestone 3 -- Testing

- **Tester (memory):** "CREATE" passes; count increases; `rec.id` is present.
- **Demo (memory):** use the `insertOne` card; confirm record appears in `getDoc()`.

CREATE (memory)


```
      "ts": "2025-10-01T15:00:00Z"
    }
  ],
  "tags": [
    {
      "id": "t-web",
      "name": "web"
    },
    {
      "id": "t-setup",
      "name": "setup"
    }
  ]
}
Finished: CREATE (memory)
```


1. ✓ insert returns record with id
2. ✓ user count increased

'Approve' → Test phase

Test Instructions:

- Serve the project over HTTP and open `/tests/test-memory.html` → run **CREATE** (should be green).
- Then open `/demos/memory.html` and run **insertOne** with the prefilled payload; use **getDoc** to confirm the new record exists.
- Want more practice? Use **Appendix A** to try additional inserts (e.g., insert users/tags) and observe that `rev/updatedAt` advance after each save.

Goal 3: Browser Database -- Read Operations

'Approach' → Plan phase

Goal:
Provide **safe, side-effect-free reads** so UIs/tests can observe state reliably. Reads must never mutate cached data and must be simple enough to reason about while remaining flexible for later enhancements.

Design for this iteration

- **Copy-on-read:** All reads go through `getDoc()` (already implemented in Goal 2) so `findMany` / `findOne` operate on a **deep copy** of `_doc`. No mutation leaks.
- **Pure selection:** `findMany(col, pred?)` returns all items matching the **predicate** (or all items if omitted). `findOne(col, pred)` returns the **first match** or `null`.
- **Simplicity > cleverness:** These are O(n) scans over the collection arrays -- perfect for lab scale and easy to reason about.
- **Guardrails later:** If `col` is missing or `pred` throws, we'll introduce friendly errors in a later "Guardrails" goal (optional).

'Apply' → Do phase

Overview of milestones

- Milestone 1: findMany
- Milestone 2: findOne

Milestone 1: findMany

Goal: Return all records in a collection that match a rule

If you come from SQL, this is like:

```
SELECT * FROM todos WHERE assignedTo = 'u-alice' AND done = false;
```

In our API, you'd write:

```
findMany("todos", t => t.assignedTo === "u-alice" && !t.done);
```

Step 1: Filter over a safe snapshot & apply the predicate function

Think of each **collection** (e.g., `"todos"`, `"users"`) as a JavaScript **array of records**. `findMany(col, pred)` returns the subset of that array that matches your rule. Here, the "rule" is a **predicate function** -- a little JS function that receives each record and returns `true` (keep it) or `false` (skip it). We always filter over a **safe snapshot** (`getDoc()`), so you can't accidentally change the database while reading.

Notes

- **Default behavior:** If you omit the predicate, `findMany(col)` returns **every record** in the collection.
- **No side effects:** Because we filter a **copy**, your predicate can't mutate the database.

`scripts/db.js` → *findMany*

```
export function findMany(col, pred = () => true) {
    return getDoc()[col].filter(pred);
}
```

Milestone 1 -- Testing

- **Tester (memory):** "READ — Find Many" block returns an array (✓).
- **Demo (memory):** run `findMany` with `assignedTo === "u-alice"` and see matching todos.

Milestone 2: findOne

Goal: Return the first record in a collection that matches a rule

If you come from SQL, this is like:

```
SELECT * FROM todos WHERE assignedTo='u-alice' AND done=false LIMIT 1;
```

In our API:

```
findOne("todos", t => t.assignedTo === "u-alice" && !t.done);
```

Step 1: First match (or null) over a safe snapshot

Think of each **collection** (`"todos"`, `"users"`, …) as a JavaScript **array of records**.
`findOne(col, pred)` scans that array and returns the **first** record for which the **predicate function** returns `true`; if none match, it returns **`null`**. We always read from a **deep copy** (`getDoc()`), so your predicate can't accidentally mutate the cached DB.

Notes

- **Default behavior:** There's no default predicate for `findOne`; you must provide a rule.
- **Determinism:** If multiple records could match, your predicate should be specific enough to pick the one you intend.
- **Performance:** Simple **O(n)** scan—great for lab-scale data. (Advanced CRUD adds filter objects if you prefer JSON-style queries.)

`scripts/db.js` → *findOne*

```
// read one
export function findOne(col, pred) {
  return getDoc()[col].find(pred) || null;
}
```

Milestone 2 -- Testing

- **Tester (memory):** "READ — Find One" finds the expected record (✓) or `null`.
- **Demo (memory):** query `users` where `name === "Alice"` and confirm the single record.

READ (memory) -- Find One

```
Running: READ (memory) -- Find One
→ findOne(users, u => u.name === "Carol")
← findOne ✓ result: {
  "id": "3ac7cc18",
  "name": "Carol"
}
Finished: READ (memory) -- Find One
```

1. ✓ findOne locates Carol

'Approve' → Test phase

Test Instructions:

- Serve over HTTP and open `/tests/test-memory.html`; verify the **READ** blocks pass (arrays for many, object/null for one).
- Open `/demos/memory.html` and try a few predicates (e.g., by `assignedTo`, `done`, or `name`).
- For more patterns, consult **Appendix A**

Goal 4: Browser Database -- Update Operations

'Approach' → Plan phase

Goal

Enable **safe, single-record updates** with predictable semantics. An update must: (1) locate a record by `id`, (2) apply a **shallow** patch (top-level fields only; **arrays are replaced**), and (3) **write through** the adapter so `rev/updatedAt` are stamped and the cache stays in sync--then return **`1`** if a record was updated, **`0`** if no match.

Design

- **Copy → mutate → save:** Always work on a **deep copy** via `getDoc()`, never mutate `_doc` directly; then `await _adapter.save(d)` and set `_doc = d`.
- **Shallow patch:** `{ ...old, ...patch }` (no deep merge); arrays (e.g., `tagIds`) are replaced—compute the full next array before calling `updateOne`.

  Need to modify array items often? Use **Advanced CRUD**'s `updateOneOps` (`$addToSet/$pull`) later.
- **Return code:** `1` when an update occurred, `0` when `id` wasn't found (idempotent & easy to test).

'Apply' → Do phase

Overview of milestones

- Milestone 1: updateOne

Milestone 1: updateOne

Goal: Update a single record in a collection by id

If you come from SQL, this is like:

```
UPDATE todos SET done = TRUE, assignedTo = 'u-bob' WHERE id = 't-1';
```

In our API:

```
await updateOne("todos", "t-1", { done: true, assignedTo: "u-bob" });
```

Step 1: Locate by id, shallow-patch, save, return 1 | 0 (single-record update)

Finds the target record, merges **top-level** fields from `patch` into it (arrays are **replaced**, not merged), then performs a **write-through** save via the adapter (which stamps `rev/updatedAt`) and refreshes the cache. It returns **`1`** when a record was updated and **`0`** if no record with that `id` exists.

Notes

- **Shallow patch:** `{ ...old, ...patch }` only; nested objects/arrays are not deep-merged.
- **Arrays replaced:** compute the **full next array** before calling (use Advanced CRUD's `$addToSet/$pull` later if you need array ops).
- **`id` is immutable:** do not include `id` in `patch`.

`scripts/db.js` → *updateOne*

```
// apply shallow patch; arrays are replaced
export async function updateOne(col, id, patch) {
    const d = getDoc();
    const i = d[col].findIndex(r => r.id === id);
    if (i === -1) return 0;
    d[col][i] = { ...d[col][i], ...patch };
    await _adapter.save(d);
    _doc = d;
    return 1;
}
```

Milestone 1 -- Testing

- **Tester (memory):** "UPDATE" block passes (returns 1; updated field visible).

UPDATE (memory)

```
  "done": true
})
← updateOne ✓ result: 1
→ findOne(todos, x => x.id === t.id)
← findOne ✓ result: {
  "id": "dc45cf68",
  "title": "Draft syllabus",
  "done": true,
  "assignedTo": "u-bob",
  "projectId": "p-1",
  "tagIds": [],
  "subtasks": [],
  "commentIds": []
}
Finished: UPDATE (memory)
```

1. ✓ updateOne returns 1
2. ✓ todo marked done

'Approve' → Test phase

Test Instructions:

- Serve over HTTP and open `/tests/test-memory.html`; verify the **UPDATE** blocks pass.
- Open `/demos/memory.html` and try a few predicates (e.g., by `assignedTo`, `done`, or `name`).
- For more patterns, consult **Appendix A**

Goal 5: Browser Database -- Delete Operations

'Approach' → Plan phase

Goal

Enable safe, single-record **deletions** with predictable semantics. A delete must: (1) **locate** a record by `id`, (2) **remove** it from the target collection, and (3) **write through** the adapter so `rev/updatedAt` are stamped and the cache stays in sync—then return `1` if a record was deleted, `0` if no match (idempotent).

Design

- **Copy → mutate → save:** Always work on a **deep copy** via `getDoc()`; never mutate `_doc` directly. After removal, `await _adapter.save(d)` and set `_doc = d`.
- **Idempotent contract:** If the `id` isn't present, do nothing and return `0`. Repeating a successful delete immediately returns `0`.
- **Narrow scope:** Only the specified **collection** changes; other collections remain untouched.
- **Stamping:** Adapters are responsible for bumping `rev` and setting `updatedAt` on successful saves (consistent across Memory / LocalStorage / JSONBin).

'Apply' → Do phase

Overview of milestones

- Milestone 1: deleteOne

Milestone 1: deleteOne

Goal: Remove a single record from a collection by id

SQL analogy:

```
DELETE FROM tags WHERE id = 't-setup';
```

Our API

```
await deleteOne("tags", "t-setup"); // → 1 if deleted, 0 if not found
```

Step 1: Filter by id, save if changed, return 1|0

Works on a **safe copy** (`getDoc()`), removes any record whose `id` matches, and, **only if the length changed**, writes through the adapter (which stamps `rev/updatedAt`) and refreshes the cache. It returns **`1`** when a row was removed and **`0`** when the `id` wasn't found (idempotent, easy to test).

`scripts/db.js` → *deleteOne*

```
export async function deleteOne(col, id) {
    const d = getDoc();
    const before = d[col].length;
    d[col] = d[col].filter(r => r.id !== id);
    const deleted = before - d[col].length;
    if (deleted) {
        await _adapter.save(d);
        _doc = d;
    }
    return deleted; // 0 or 1
}
```

Milestone 1 -- Testing

- **Tester (memory):** "DELETE" passes when `deleteOne` returns `1` for a valid `id` and removes the record; a second call with the same `id` returns `0`.

DELETE (memory)

```
Running: DELETE (memory)
→ insertOne(tags, {
  "name": "temp-tag"
})
← insertOne ✓ result: {
  "id": "82852b97",
  "name": "temp-tag"
}
→ deleteOne(tags, 82852b97)
← deleteOne ✓ result: 1
→ findOne(tags, x => x.id === tag.id)
← findOne ✓ result: null
Finished: DELETE (memory)
```

1. ✓ deleteOne returns 1
2. ✓ record removed

'Approve' → Test phase

Test Instructions:

- Serve over HTTP and open `/tests/test-memory.html`; verify the **DELETE** blocks pass.
- Open `/demos/memory.html` and try a few predicates (e.g., by `assignedTo`, `done`, or `name`).
- For more patterns, consult **Appendix A**

Dev Cycle 2:
Advanced CRUD operations

What you're building (and why)

In this cycle, you'll extend the browser database with **Mongo-like query features** -- so you can **query, shape, and modify** records more expressively. This is still **in-memory** (volatile) for fast iteration; persistence arrives in the next cycles.

Objectives (end-of-cycle checklist)

By the end of Cycle 2, your DB will support:

- **Filter objects** — e.g., `{ assignedTo:"u-alice", done:false }`, plus `$in`, `$gt/$gte`, `$lt/$lte`, `$ne`, `$contains`.
- **Projection / Sort / Paging** — return only selected fields; order results; support `skip`/`limit`.
- **Update operators** — sugar for arrays and fields: `$set`, `$addToSet`, `$pull`.
- **Upsert** — create the record if it doesn't exist; otherwise, update the existing match.

Result:

a **more capable document DB** that can power serverless browser apps and prepares you to map the same patterns to MongoDB later.

Table of Contents

Goal 1: Browser Database -- Filter Operations

'Approach' → Plan phase

Goal

Let callers use **JSON-style filter objects** (Mongo-ish) instead of writing JS predicate functions. This makes queries easier to read, share, and auto-generate from UI controls.

Design

- **Copy-on-read:** Evaluate filters against `getDoc()` (deep copy), so reads never mutate `_doc`.
- **AND semantics:** A filter is an **AND** of its fields (all must match).
- **Supported operators (per field):** direct equality, plus `$in`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$contains`.
- **Scale:** Simple **O(n)** scans are acceptable at this lab scale.

SQL analogy

```
SELECT * FROM todos
WHERE assignedTo = 'u-alice' AND done = false AND due >= '2025-10-01';
```

In our API (filter object instead of a predicate):

```
queryBy(todo, {
    assignedTo: "u-alice",
    done: false,
    due: { $gte: "2025-10-01" }
});
```

'Apply' → Do phase

Overview of milestones

- Milestone 0: Define the query commands for our mock database
- Milestone 1: findManyBy(col, filter) — array of matches
- Milestone 2: findOneBy(col, filter) — first match (or null)

Milestone 0: Define the query commands for our mock database

Step 0: queryBy(row, filter) — evaluate one record against a filter

What this does: Given a single record (`row`) and a filter object, return `true` if the record matches **all** filter clauses.

Supported operators (per field)

- **Direct equality:** `{ field: value }` → `field = value`
- **`$in`:** `{ field: { $in: [v1, v2, …] } }` → `field IN (v1, v2, …)`
- **`$gt` / `$gte`:** `>` / `>=` (numeric, string, or date strings as-is)
- **`$lt` / `$lte`:** `<` / `<=`
- **`$ne`:** `!=`
- **`$contains`:** substring test (case-sensitive): `{ field: { $contains: "foo" } }`

AND semantics: `{ a:1, b:2 }` means `a == 1 AND b == 2`.
No side effects: Filtering never writes to your data.

scripts/db.js → *queryBy*

```
// very small filter interpreter (eq/neq/gt/gte/lt/lte/in/contains)
export function queryBy(row, filter = {}) {
  return Object.entries(filter).every(([k, v]) => {
    if (v && typeof v === "object" && !Array.isArray(v)) {
      if ("$in" in v)   return v.$in.includes(row[k]);
      if ("$gt" in v)   return row[k] >  v.$gt;
      if ("$gte" in v)  return row[k] >= v.$gte;
      if ("$lt" in v)   return row[k] <  v.$lt;
      if ("$lte" in v)  return row[k] <= v.$lte;
      if ("$ne" in v)   return row[k] !== v.$ne;
      if ("$contains" in v) return String(row[k] ?? "").includes(String(v.$contains));
    }
    return row[k] === v;
  });
}
```

Examples

```
// AND across fields
queryBy(t, { assignedTo: "u-alice", done: false });

// Comparators / set membership
queryBy(t, { due: { $gte: "2025-10-01" } });
queryBy(u, { id:  { $in: ["u-alice","u-bob"] } });

// String contains
queryBy(p, { name: { $contains: "Course" } });
```

Milestone 0 — Testing

- **No standalone tests;** this helper powers the next milestones.

Milestone 1: findManyBy(col, filter) — array of matches

Goal

Return all records in a collection that match a JSON-style filter.

SQL analogy:

```
SELECT * FROM todos WHERE assignedTo='u-alice' AND done=false;
```

In our API (filter object instead of a predicate):

```
findManyBy("todos", { assignedTo: "u-alice", done: false });
```

Step 1: Filter object over a safe snapshot (AND across fields)

Treat each collection (`"todos"`, `"users"`, …) as a JavaScript array of records.
`findManyBy(col, filter)` evaluates the **filter object** against a **deep copy** of the collection (via `getDoc()`), keeping any record where **all** filter fields match. Per-field operators supported: direct equality, plus `$in`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, and `$contains` (string contains).

Notes

- **AND semantics:** `{ a:1, b:2 }` means `a==1 AND b==2`.
- **No side effects:** Reads use a **copy**, so you can't mutate the DB while filtering.
- **Lab-scale performance:** Simple **O(n)** scans—perfect for our data sizes.

.

`scripts/db.js` → *findManyBy*

```
export function findManyBy(col, filter = {}) {
    return getDoc()[col].filter(r => queryBy(r, filter));
}
```

Examples of Usage

```
findManyBy("todos", { assignedTo: "u-alice", done: false });
findManyBy("tags",  { name: { $contains: "up" } });  // matches "setup"
findManyBy("users", { id: { $in: ["u-alice", "u-bob"] } });
```

Milestone 1 — Testing

- **Tester (memory):** "Advanced — Filter Objects (findManyBy)" returns an **array** (✓).
- **Demo (memory):** Paste `{ "col":"todos", "filter": { "assignedTo":"u-alice", "done":false} }` into the **findManyBy** card and run; expect matching todos.

Milestone 2: findOneBy(col, filter) — first match (or null)

Goal

Return the **first** record in a collection that matches a JSON-style filter.

SQL Analogy:

```
SELECT * FROM todos
WHERE assignedTo='u-alice' AND done=false
LIMIT 1;
```

In our API (filter object instead of a predicate)

```
findOneBy("todos", { assignedTo: "u-alice", done: false });
```

Step 1: Filter object over a safe snapshot; return first match or `null`

Treat each collection (`"todos"`, `"users"`, …) as a JavaScript array of records.
`findOneBy(col, filter)` evaluates the **filter object** against a **deep copy** of the collection (via `getDoc()`), returning the **first** record where **all** filter fields match—or **`null`** if none match. Supported per-field operators: direct equality, `$in`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, and `$contains` (string contains).

Notes

- **AND semantics:** `{ a:1, b:2 }` means `a == 1 AND b == 2`.
- **No side effects:** Reads operate on a **copy**, so filters can't mutate the DB.
- **Determinism:** If multiple rows could match, your filter should be specific enough to select the intended one.
- **Lab-scale performance:** Simple **O(n)** scan—acceptable for our data sizes.

`scripts/db.js` → *findOneBy*

```
export function findOneBy(col, filter = {}) {
    return getDoc()[col].find(r => queryBy(r, filter)) ?? null;
}
```

Examples of Usage

```
findOneBy("tags",  { name: "setup" });                                  // seed match
findOneBy("users", { id: { $in: ["u-alice", "u-bob"] } });              // first of the two
findOneBy("todos", { due: { $gte: "2025-10-10" }, done:false });
findOneBy("projects", { name: { $contains: "Course" } });
findOneBy("todos", { id: "no-such-id" });                               // null;
```

Milestone 1 — Testing

- **Tester (memory):** "Advanced — Filter Objects (findOneBy)" returns the **expected record** (or `null` when no match).
- **Demo (memory):** Paste `{ "col":"tags", "filter": { "name":"setup" } }` into the **findOneBy** card and run; expect the tag object. .

Approve' → Test phase

Test Instructions:

- Open the **launcher** at `/index.html` and click:
- **Test — In-Memory (Part 2)** to run advanced filter tests (findManyBy / findOneBy / find / updateOneOps / upsert / transact).
- **Demo — In-Memory (Part 2)** to try JSON payloads interactively (same operations, live console).
- **Passing** for this goal: the advanced tester blocks are green (especially **Filter Objects**), and the demo returns an **array** for `findManyBy` and a **record or null** for `findOneBy` with no errors.
  (Tester page and demo referenced above: **test-adv-memory.html**, **memory-adv.html**.)
- If anything fails, re-open the demo, paste the same filter you used in the test, and compare outputs against the **Cheat Sheet** (operators: equality, `$in`, `$gt/$gte`, `$lt/$lte`, `$ne`, `$contains`).
- For more patterns, consult **Appendix A**

Goal 2: Browser Database - Projection, Sort, Skip/Limit

'Approach' → Plan phase

Goal

Support common **list-view ergonomics**: shape the fields you return (**projection**), control **ordering** (**sort**), and page through results (**skip/limit**). These features sit on top of your **filter** (from Goal 1) and help you build real list UIs cleanly.

Design

- **Copy-on-read:** Always operate on the **deep copy** from `getDoc()`; reads never mutate `_doc`.
- **Pipeline order:** `filter` → `sort` → `skip` → `limit` → `project`.
- **Scale:** Straightforward **O(n)** scans and in-memory sorts are acceptable at lab scale.
- **Projection syntax:** `fields` is an **include map** (e.g., `{ id:1, title:1 }` keeps only those keys).

In SQL:

```
SELECT id, title
FROM   todos
WHERE  done = false
ORDER  BY title ASC
OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY;
```

In our API (filter object instead of a predicate)

```
find("todos", {
  filter: { done: false },
  sort:   { title: 1 },     // 1 = ASC, -1 = DESC
  skip:   0,
  limit:  3,
  fields: { id: 1, title: 1 }
});
```

'Apply' → Do phase

Overview of milestones

- **Milestone 0:** `project(row, fields)` — shape a single record
- **Milestone 1:** `find(col, args)` — filter + sort + skip/limit + projection

Milestone 0: `project(row, fields)` — shape a single record

Goal

Return only the requested fields from a record (or the record untouched if no `fields` map is provided).

Step 1: Include-only projection map

Keep keys where `fields[key]` is truthy; ignore all others.

`scripts/db.js` → *project*

```
export function project(row, fields) {
  if (!fields) return row;
  const out = {};
  for (const k of Object.keys(fields)) if (fields[k]) out[k] = row[k];
  return out;
}
```

Milestone 0 — Testing

- **No standalone tests;** `project` is exercised through `find()` in the next milestone

Milestone 1: `find(col, args)` — one call for list ergonomics

Goal

Return a list with **filter**, optional **sort**, **skip/limit**, and optional **projection**.

Step 1: Base (wrap filter)

Start with `findManyBy(col, filter)`.

scripts/db.js → *find*

```
export function find(col, { filter={} }={}) {
  let rows = findManyBy(col, filter);
  return rows;
}
```

Step 2: Add sort

Single-key sort: `{ title: 1 }` (ASC) or `{ title: -1 }` (DESC).

scripts/db.js → *find*

```
export function find(col, { filter={}, sort=null }={}) {
  let rows = findManyBy(col, filter);
  if (sort) {
    const [[k, dir]] = Object.entries(sort);
    rows = rows.slice().sort((a,b) => (a[k] > b[k] ? 1 : a[k] < b[k] ? -1 : 0) * (dir === -1 ? -1 : 1));
  }
  return rows;
}
```

Step 3: Add limit

Keep only the first `limit` rows (if provided).

scripts/db.js → *find*

```
export function find(col, { filter={}, sort=null, limit=null }={}) {
  let rows = findManyBy(col, filter);
  if (sort) {
    const [[k, dir]] = Object.entries(sort);
    rows = rows.slice().sort((a,b) => (a[k] > b[k] ? 1 : a[k] < b[k] ? -1 : 0) * (dir === -1 ? -1 : 1));
  }
  if (limit) rows = rows.slice(0, limit);
  return rows;
}
```

Step 4: Add skip

Skip the first `skip` rows (defaults to 0).

scripts/db.js → *find*

```
export function find(col, { filter={}, sort=null, limit=null, skip=0 }={}) {
  let rows = findManyBy(col, filter);
```

```
  if (sort) {
    const [[k, dir]] = Object.entries(sort);
    rows = rows.slice().sort((a,b) => (a[k] > b[k] ? 1 : a[k] < b[k] ? -1 : 0) * (dir === -1 ? -1 : 1));
  }
  if (skip) rows = rows.slice(skip);
  if (limit) rows = rows.slice(0, limit);
  return rows;
}
```

Step 5: Add projection (`fields`)

Return only the requested fields per row.

scripts/db.js → find

```
export function find(col, { filter={}, sort=null, limit=null, skip=0, fields=null }={}) {
  let rows = findManyBy(col, filter);
  if (sort) {
    const [[k, dir]] = Object.entries(sort);
    rows = rows.slice().sort((a,b) => (a[k] > b[k] ? 1 : a[k] < b[k] ? -1 : 0) * (dir === -1 ? -1 : 1));
  }
  if (skip) rows = rows.slice(skip);
  if (limit) rows = rows.slice(0, limit);
  if (fields) rows = rows.map(r => project(r, fields));
  return rows;
}
```

Milestone 1 — Testing

- **Tester (memory):** The “Advanced — Projection / Sort / Paging (find)” block returns an **array**, respects `limit`, and applies `fields`.

- **Demo (memory):** Use the **find** card with `{ "filter":{"done":false}, "sort":{"title":1}, "limit":3, "fields":{"id":1,"title":1} }`; verify ordering, count, and fields.

'Approve' → Test phase

Test Instructions:

Open `/index.html` and click:

- **Test — In-Memory (Part 2)** to run **Projection / Sort / Paging** tests (green = pass).
- **Demo — In-Memory (Part 2)** to try `find` interactively (paste the examples above).
- For more patterns, consult **Appendix A**

Passing for this goal: the advanced tester's **find** block is green; the demo shows sorted, paged results with the **projected fields** only.

If anything fails, compare your `find` pipeline (filter → sort → skip → limit → project) with the **Cheat Sheet** examples and adjust.

Goal 3: Browser Database - Advanced Update Operator

'Approach' → Plan phase

Goal
Add a **Mongo-like operator helper** so your app can update records concisely -- especially arrays -- without hand-building full replacement arrays each time.

Design

- **Keep `updateOne` as the core**: operators merely **compile into a shallow patch**; `updateOne(col,id,patch)` still does the write-through and cache refresh.

- **Operators → patch**
  - **`$set`** — shallow set of top-level fields.
  - **`$addToSet`** — add unique item(s) to an array (no duplicates).
  - **`$pull`** — remove a value from an array.

- **Safety**: Arrays are still **replaced** (we compute full next arrays); `id` remains immutable; adapter stamps `rev/updatedAt`.

In SQL

```
` -- SET done = TRUE
UPDATE todos
SET    done = TRUE
WHERE  id = 't-1';

-- ADD tag 't-web' only if it's not already present (add-to-set)
INSERT INTO todo_tags (todo_id, tag_id)
SELECT 't-1', 't-web'
WHERE NOT EXISTS (
  SELECT 1
  FROM   todo_tags
  WHERE  todo_id = 't-1' AND tag_id = 't-web'
);
```

In our API (patch object instead of a predicate)

```
await updateOneOps("todos", "t-1", {
   $set:      { done: true },
   $addToSet: { tagIds: "t-web" }
});
```

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** `updateOneOps` — translate `{ $set, $addToSet, $pull }` into a shallow patch

Milestone 1: updateOneOps

Goal

Provide a convenience wrapper around `updateOne` that accepts a **compact operator object** and performs array-safe updates.

Step 1: base (wrap updateOne)

Provide the **scaffold**: read the row, build a patch object, and call `updateOne`.

- **When to use:** You want a single entry point that compiles operator objects to a shallow patch, then calls `updateOne`.
- *(No SQL mapping yet—this step is just the scaffold.)*

`scripts/db.js` → *updateOneOps*

```
export async function updateOneOps(col, id, ops = {}) {
  const row = findOne(col, r => r.id === id);
  if (!row) return 0;
  const patch = {};
  return updateOne(col, id, patch);
}
```

Step 2: add $set

Allow shallow setting of top-level fields via `{ $set: { field: value, ... } }`.

- **When to use:** Set/overwrite top-level scalar fields (e.g., `done`, `title`, `assignedTo`).
- **SQL analogy:**

```
UPDATE todos SET done = TRUE WHERE id = 't-1';
```

`scripts/db.js` → *updateOneOps*

```
export async function updateOneOps(col, id, ops = {}) {
  const row = findOne(col, r => r.id === id);
  if (!row) return 0;
  const patch = {};
  if (ops.$set) Object.assign(patch, ops.$set);
  return updateOne(col, id, patch);
}
```

Step 3: add $addToSet

Append unique value(s) to array field(s) without duplicates.

- **When to use:** Append a value to an array field **without duplicates** (e.g., add a tag to `tagIds`).
- **SQL analogy (portable join-table)**

```
INSERT INTO todo_tags (todo_id, tag_id)
SELECT 't-1', 't-web'
WHERE NOT EXISTS (
   SELECT 1 FROM todo_tags WHERE todo_id='t-1' AND tag_id='t-web'
);
```

scripts/db.js → *updateOneOps*

```
export async function updateOneOps(col, id, ops = {}) {
  const row = findOne(col, r => r.id === id);
  if (!row) return 0;
  const patch = {};
  if (ops.$set) Object.assign(patch, ops.$set);
  if (ops.$addToSet) {
    for (const [k, v] of Object.entries(ops.$addToSet)) {
      const cur = Array.isArray(row[k]) ? row[k] : [];
      patch[k] = Array.from(new Set([...cur, ...(Array.isArray(v) ? v : [v])]));
    }
  }
  return updateOne(col, id, patch);
}
```

Step 4: add $pull

Remove a value from array field(s).

- **When to use:** Remove a value from an array field (e.g., remove a tag).
- **SQL analogy (portable join-table):**

```
DELETE FROM todo_tags WHERE todo_id = 't-1' AND tag_id = 't-web';
```

scripts/db.js → *updateOneOps*

```
export async function updateOneOps(col, id, ops = {}) {
  const row = findOne(col, r => r.id === id);
  if (!row) return 0;
  const patch = {};
  if (ops.$set) Object.assign(patch, ops.$set);
  if (ops.$addToSet) {
    for (const [k, v] of Object.entries(ops.$addToSet)) {
      const cur = Array.isArray(row[k]) ? row[k] : [];
      patch[k] = Array.from(new Set([...cur, ...(Array.isArray(v) ? v : [v])]));
    }
  }
  if (ops.$pull) {
    for (const [k, v] of Object.entries(ops.$pull)) {
      const cur = Array.isArray(row[k]) ? row[k] : [];
      patch[k] = cur.filter(x => x !== v);
    }
  }
  return updateOne(col, id, patch);
}
```

Milestone 1 — Testing

Examples of Usage

```
// Set a scalar and add a tag (no duplicates)
await updateOneOps("todos", "t-1", { $set: { done: true }, $addToSet: { tagIds: "t-web" } });

// Pull the tag back out
await updateOneOps("todos", "t-1", { $pull: { tagIds: "t-web" } });

// Rename a user
await updateOneOps("users", "u-bob", { $set: { name: "Robert" } });
```

Notes

- Operators only produce a **shallow** patch; nested objects/arrays are not deep-merged.
- Arrays are **replaced** with the computed next arrays (consistent with `updateOne`).
- Return value mirrors `updateOne`: `1` when updated, `0` when no row matched.

'Approve' → Test phase

Test Instructions:

Open `/index.html` → click **Test — In-Memory (Part 2)** and verify the **Advanced — updateOneOps ($set / $addToSet / $pull)** block is **green**.

Open `/demos/demo-adv-memory.html` and run the **updateOneOps** card:

- Add a tag with `$addToSet`, flip `done` with `$set`, then remove the tag with `$pull`.
- Confirm `getDoc()` shows the intended changes and that `rev/updatedAt` advanced.
- For more patterns, consult **Appendix A**

Goal 4: Browser Database - Upsert *(Create-or-Update)*

'Approach' → Plan phase

Goal
Provide a **create-or-update convenience** so callers can “ensure this record exists” without writing the lookup flow every time.

Design

- **Find-then-act:** Use `findOneBy(col, filter)` to locate a row. If found, **update** it (then re-read and return). If not found, **insert** a new record and return it.
- **Return value:** Always return the **resulting record** (updated or newly inserted), not `0/1`.
- **Safety:** Upsert does *not* modify `id` and still relies on `updateOne` / `insertOne` for write-through and stamping (`rev/updatedAt`).

SQL

```
-- Ensure there's exactly one 'priority' row in tags
-- 1) Update if it exists (no-op here since we're setting the same value)
UPDATE tags
SET    name = 'priority'
WHERE  name = 'priority';

-- 2) Insert if it doesn't exist
INSERT INTO tags (name)
SELECT 'priority'
WHERE  NOT EXISTS (SELECT 1 FROM tags WHERE name = 'priority');
```

Our API

```
await upsertOne("tags", { name: "priority" }, { name: "priority" });
```

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** `upsertOne` — update a record if it exists, otherwise create it

Milestone 1: upsertOne

Goal:

Implement “create or update” using **find → update-or-insert → return the record**.

Step 1: Try findOneBy; update if found, else insert

Upsert: Update the first matching record (by filter) or insert a new record. Returns the updated/inserted record (not 0/1).

`scripts/db.js` → *upsert*

```
export async function upsertOne(col, filter, data) {
  const existing = findOneBy(col, filter);

  if (existing) {
    // Update the found record, then re-read and return it
    await updateOne(col, existing.id, data);
    return findOne(col, r => r.id === existing.id);
  }
  else {
    // No match → insert and return the new record
    return insertOne(col, data);
  }
}
```

Examples of Usage

```
// Ensure tag "priority" exists (first call inserts, later calls update)
await upsertOne("tags", { name: "priority" }, { name: "priority" });

// Ensure a user record shape (will update if filter matches an existing user)
await upsertOne("users", { name: "Carol" }, { name: "Carol" });
```

Notes

- **Idempotent intent:** Repeated calls with the same filter/data eventually converge to the same record.
- **Race awareness:** In rare cases (record disappears between find/update), `updateOne` could no-op—if needed, handle it by checking the return code and falling back to `insertOne`.
- **Composability:** Pair `upsertOne` with `updateOneOps` when you also need `$addToSet/$pull` array updates.

Milestone 1 — Testing

- **Tester (memory):** The “Advanced — Upsert section should pass with a green check..

'Approve' → Test phase

Test Instructions:

- Open `/index.html` → click **Test — In-Memory (Part 2)** and verify the upsert block passes (first call **creates**, second call **updates** without duplicates).
- Open `/demos/demo-adv-memory.html` and run the **upsertOne** card with `{ "col":"tags", "filter":{"name":"priority"}, "data":{"name":"priority"} }`; confirm the tag appears on first run and doesn't duplicate on subsequent runs.
- **Passing:** the tester's upsert assertions are green and the demo shows the same record returned on repeated calls.

Goal 5: Browser Database - Batch (Transact)

'Approach' → Plan phase

Goal
Provide a simple **batch edit** so you can make multiple changes across one or more collections and **save once**. This reduces redundant writes, keeps related edits together, and mirrors the idea of an atomic "transaction" (at our lab scale).

Design

- **Copy → mutate → save (one time):** Start from a **deep copy** (`getDoc()`), let the caller mutate that copy, then `await _adapter.save(d)` and set `_doc = d`.
- **Single save = single stamp:** The adapter stamps `rev/updatedAt` **once** for the whole batch.
- **Mutator function:** Accepts a function `mutatorFn(doc)` (sync **or** async) that performs all related changes (add, update, delete) on the working copy.
- **Scope & safety:** This is not a DB-level ACID transaction—it's a single in-memory mutation followed by one save (perfect for our document-store lab).

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** `transact(mutatorFn)`

Milestone 1: `transact(mutatorFn)`

Goal

Run a user-supplied function to apply **one or more edits** to the AppDoc, then **persist once**.

Step 1: Copy, run mutator, save once, refresh cache

Create a **safe working copy** of the AppDoc with `getDoc()`, then run the caller's `mutatorFn(d)` to apply **any number of edits** to that copy (across one or more collections). Persist **once** via `_adapter.save(d)` (the adapter stamps `rev/updatedAt`), then set `_doc = d` so the in-memory cache reflects the new canonical state and return it. This gives you a simple, one-save "batch" update path without mutating the live cache mid-operation.

`scripts/db.js` → *transact*

```
export async function transact(mutatorFn) {
  const d = getDoc();
  mutatorFn(d);
  await _adapter.save(d);            // single write-through; adapter stamps rev/updatedAt
  _doc = d;                          // refresh cache
  return _doc;                       // return the new canonical doc
}
```

Example Usage ***(do not use this code -- it's an example only)***

```
// Ensure tag 't-priority' exists and attach it to the first todo
await transact(doc => {
  if (!doc.tags.find(t => t.id === "t-priority"))
    doc.tags.push({ id: "t-priority", name: "priority" });
  const first = doc.todos[0];
  if (first) {
    const set = new Set([...(first.tagIds || []), "t-priority"]);
    first.tagIds = Array.from(set);
  }
});

// Rename a project and retag all its todos in one save
await transact(doc => {
  const p = doc.projects.find(x => x.id === "p-1");
  if (p) p.name = "Course Platform";
  for (const t of doc.todos.filter(t => t.projectId === "p-1")) {
    t.tagIds = Array.from(new Set([...(t.tagIds || []), "t-web"]));
  }
});
```

Milestone 1 — Testing

- **Tester (memory):** The "Advanced — Upsert section should pass with a green check.

'Assess' → Test phase

Test Instructions:

- **Tester:** open `/index.html` → **Test — In-Memory (Part 2)** → ensure the **"Advanced — transact (batch save)"** block is **green**.
- **Demo:** open `/demos/demo-adv-memory.html` → use the **transact** card (e.g., add a tag and attach it to the first todo).
    - **Pass when:** the demo shows the tag added and attached in a single run, and `rev/updatedAt` advance **once**.
- If something fails, verify your flow is strictly `getDoc()` → **mutate copy** → `_adapter.save(d)` → `_doc = d` with no intermediate saves.

Dev Cycle 3:
LocalStorage Adapter

What you're building (and why)

In this cycle, you'll add **persistence** to your browser database by swapping the in-memory adapter for a **LocalStorage adapter**. The DB API stays the same; only the storage engine changes. This lets data **survive page reloads** -- ideal for serverless / browser-only apps.

What is LocalStorage (Browser API)

LocalStorage is a **built-in key/value store** in the browser that **persists across page reloads** (per origin). It stores **strings only**, is **synchronous**, and typically offers **~5–10 MB** per origin. You'll **serialize** your AppDoc with `JSON.stringify` and **deserialize** with `JSON.parse`. It's great for simple, client-only persistence; it's not a database server (no queries/indexes/transactions).

Objectives (end-of-cycle checklist)

By the end of Cycle 3, your DB will:

- Load the AppDoc from **LocalStorage** (or **seed** it on first run).
- **Save** changes with a single call (write-through) and **stamp** `rev/updatedAt`.
- **Reset** stored state for a clean slate; **snapshot** current storage for debugging.
- Pass the **basic + advanced** tests with persistence verified across reboot.

Adapter Contract (what each adapter must implement)

Your DB module calls these four methods. If an adapter implements them, it will "just work" with the existing DB API (CRUD, advanced ops, transact).

```
interface Adapter {
  // Load the latest AppDoc (or seed if missing/corrupt). Must not mutate _doc in DB.
  load(): Promise<AppDoc>;  // may be sync for LocalStorage, but treat as async

  // Persist the provided AppDoc. Should stamp next.rev/next.updatedAt (if stampOnSave).
  save(next: AppDoc): Promise<void>;

  // Optional: clear persisted state so the next load() reseeds.
  reset(): void | Promise<void>;

  // Optional: return a deep copy of the currently stored doc (or null if none).
  snapshot(): AppDoc | null | Promise<AppDoc | null>;
}
```

Invariants & responsibilities

- **Adapter stamps**: On successful `save(next)`, bump **`rev`** and set **`updatedAt`** (UTC ISO).
- **No mutation leaks**: `load()` returns the stored doc; the DB layer will clone on read (`getDoc()`), and your demo/tests will treat reads as immutable.
- **Reseed behavior**: If storage is empty or corrupt, `load()` should **seed** from `seedDoc()` and return that.
- **Async-friendly**: Even if LocalStorage is synchronous, keep the `load/save` signatures `async` for cross-adapter compatibility.

How the DB uses the contract

- `useAdapter(adapter)` — select storage engine.
- `boot()` — calls `adapter.load()`, caches the doc as the **single source of truth**.
- `insertOne/updateOne/deleteOne/transact` — modify a **working copy**, then `await adapter.save(next)` and update the cache.

Acceptance checklist for Cycle 3

- `load()` returns seed on first run and real data thereafter.
- `save()` persists and **stamps** `rev/updatedAt`.
- `reset()` clears storage; the next `load()` reseeds.
- `snapshot()` shows what's in storage (read-only).
- All **basic + advanced** tests pass, and data survives a page refresh.

Table of Contents - Dev Cycle 3

Goal 1: LocalStorage Adapter

'Approach' → Plan phase

Goal

Implement a **persistent adapter** with the same contract for all adapter types into the database module: `load()`, `save(next)`, `reset()`, `snapshot()`—so the DB API doesn't change when you swap storage engines.

Design

- **Keyed storage:** Use a configurable LocalStorage key (default `"mockdb:doc"`).
- **Seeding:** If nothing is stored—or stored JSON is corrupt—**seed** via `seedDoc()` and save.
- **Stamping:** On `save(next)`, optionally bump **`rev`** and set **`updatedAt`** (`stampOnSave: true` by default) to keep version/last-modified consistent.
- **Safety:** The adapter owns persistence; the DB layer still follows **copy** → **mutate** → **save** → **refresh cache**.
- **Limitations to keep in mind:**
    - **Sync API** (avoid huge writes in hot loops)
    - **String-only** (always stringify/parse)
    - **Quota** errors possible on very large docs
    - **Per-origin** visibility (shared by all pages on the same origin)
    - **No security** for secrets—this lab's data should be non-sensitive.

Class-based adapters (power & flexibility)

Instantiable by design: Classes make it trivial to create many independent adapters -- e.g., two LocalStorage adapters with different keys, or several JSONBin adapters with different bin IDs -- so you can context-switch or run them in parallel (local + cloud) without changing DB code.

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** LocalStorageAdapter

Milestone 1: LocalStorageAdapter

Goal:

Implement a **persistent adapter** for the DB that reads/writes the AppDoc to **LocalStorage** under a configurable key and **stamps** `rev/updatedAt` on every save. The adapter must implement the shared **contract**: `load()`, `save(next)`, `reset()`, `snapshot()`. This lets you swap Memory ↔ LocalStorage (↔ JSONBin later) without changing any DB or CRUD code.

Step 1: Class definition (private fields)

.Class definition (private fields)reate a class that remembers **which LocalStorage key** to use and whether to **stamp on save**.

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter*

```
// Persistent adapter (LocalStorage) - Same contract: load(), save(next), reset(), snapshot()

import { seedDoc } from "../model.js";

export class LocalStorageAdapter {
  #key;
  #stampOnSave;
}

// Default instance (matches prior export style)
export const localStorageAdapter = new LocalStorageAdapter();
```

Step 2: Constructor & fields (key + stamp behavior)

.Initialize the key and stamping policy; bind public methods so they're safe to pass around (no `this` pitfalls).

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter.contructor*

```
constructor({ key = "mockdb:doc", stampOnSave = true } = {}) {
    this.#key = key;
    this.#stampOnSave = stampOnSave;
    this.load = this.load.bind(this);
    this.save = this.save.bind(this);
    this.reset = this.reset.bind(this);
    this.snapshot = this.snapshot.bind(this);
}
```

Step 3: `_stamp(d)` -- Standardize revision/last-modified

Increment `d.rev` (or init from 0) and set `d.updatedAt = new Date().toISOString()`. This keeps versioning consistent across adapters on every successful save.

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter._stamp*

```
#stamp(d) {
    d.rev = (d.rev ?? 0) + 1;
    d.updatedAt = new Date().toISOString();
}
```

Step 4: `_seedAndSave()` -- First-run bootstrap

When storage is empty or unreadable, generate a **seed** AppDoc and persist it under the configured key; return the seeded doc..

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter._seedAndSave*

```
#seedAndSave() {
    const d = seedDoc();
    localStorage.setItem(this.#key, JSON.stringify(d));
    return d;
}
```

Step 5: `load()` -- Load from LocalStorage (or seed if missing/corrupt)

Try to parse the stored value for `#key`; if missing or corrupt, **reseed** and return the fresh document..

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter.load*

```
async load() {
    try {
      const raw = localStorage.getItem(this.#key);
      return raw ? JSON.parse(raw) : this.#seedAndSave();
    } catch {
      // Corrupt JSON or inaccessible storage → reseed
      return this.#seedAndSave();
    }
}
```

Step 6: `save(next)` -- Persist + (optional) stamp

Optionally **stamp** `next` (`rev`, `updatedAt`) then serialize and write to LocalStorage; surface storage/quota errors clearly.

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter.save*

```
async save(next) {
    if (this.#stampOnSave) this.#stamp(next);
    localStorage.setItem(this.#key, JSON.stringify(next));
}
```

Step 7: `reset()` -- Clear persisted state

Remove the stored document so the **next** `load()` call reseeds (useful for tests or “Reset Data” in the UI).

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter.reset*

```
// Remove the stored document (next load() will reseed)
reset() {
    localStorage.removeItem(this.#key);
}
```

Step 8: `snapshot()` -- Inspect current persisted doc (read-only)

Return a **deep copy** of the current stored document, or `null` if absent; does not mutate or seed.

`scripts/adapters/localStorageAdapter.js` → *LocalStorageAdapter.snapshot*

```
// Return a safe copy of the currently stored doc, or null if none
snapshot() {
    const raw = localStorage.getItem(this.#key);
    return raw ? JSON.parse(raw) : null;
}
```

Milestone 1 — Testing

- **Tester (memory):** The “Advanced — Upsert section should pass with a green check.

'Approve' → Test phase

Test Instructions:

Open **`/index.html`** and click **Test — LocalStorage Adapter**.
Pass when: Boot & Seed is green; **CREATE/READ/UPDATE/DELETE** pass; advanced tests (filters, find/projection/sort/paging, ops, upsert, transact) pass; **reboot/persist** checks are green.

Open **`/demos/local.html`** (or **`/demos/local.html`**) and:

- Insert a record → **refresh** → confirm it persists.
- Run `updateOneOps` (`$set/$addToSet/$pull`) and `transact`; confirm `rev/updatedAt` advance on each save.

If something fails: confirm you’re serving over **HTTP** (not `file://`), the adapter is selected with **`useAdapter(new LocalStorageAdapter({ key }) )`**, and `load()` reseeds when storage is empty/corrupt.

Dev Cycle 4:
JsonBin Adapter

Part 1: JsonBin Adapter

Will expand the browser database to cloud storage via a jsonbin adapter. This version provides data persistence between all clients.

The development cycle aims to create an enhanced document database that can serve as the data layer for browser applications using public bins and no secrets or api keys.

Your Objective for Part 4

By the end of this section, you will have a simple but operational browser database. The final result will include:

- tbd

Table of Contents

Goal 0: JsonBin API

SIMPLE & ROBUST JSON STORAGE SOLUTION

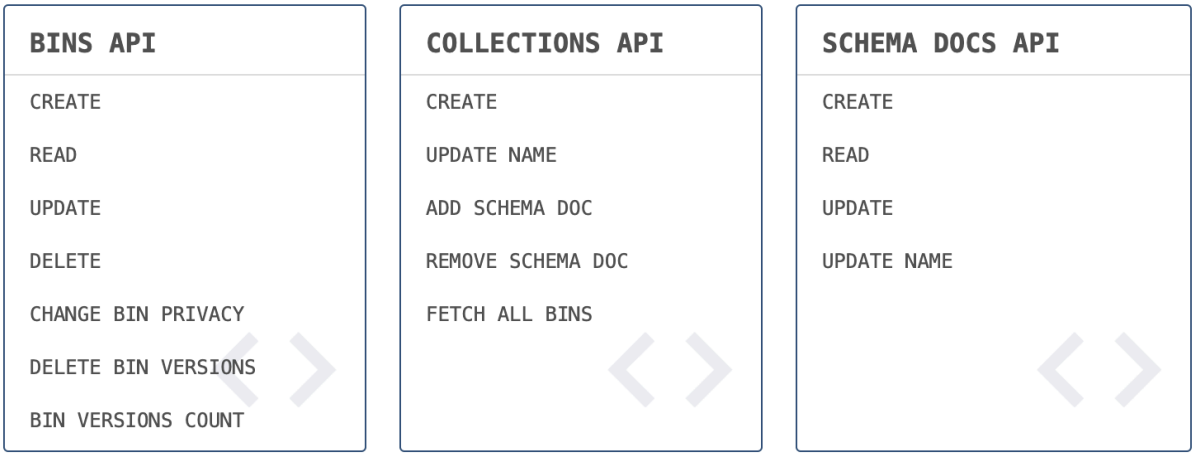
JSONBin.io provides a simple REST interface to store & retrieve your JSON data from the cloud. It helps developers focus more on the app development by taking care of their **Database Infrastructure.**

Summary:

JsonBin provides a free JSON hosting service for public or private data. Note that any data available from the browser should be public. Never use private keys in your browser code.

More Info: https://jsonbin.io/api-reference

API Overview:



- **Bins API:** Create (POST), Read (GET), Update (PUT) Private & Public bins using the Create API.
  - GET & PUT actions do not require a key, safe to perform in public code
  - POST & DELETE actions require a key, unsafe to perform in public code
- **Collections API:** Using the COLLECTIONS CREATE API, you can CREATE Collections to group the records which later, can be fetched using the Query Builder.

Goal 1: JsonBin -- Make Public BIN + Create Document

'Approach' → Plan phase

Goal

Provision a **PUBLIC JSONBin** and seed it with your app's **empty document** so the browser can do **client-only** persistence via **GET /latest** and **PUT**—no API keys in the code, no server required. (JSONBin allows GET/PUT without keys, but **POST/DELETE require keys**, so creation happens in the dashboard, not in the browser.)

Why this works for client-only apps

- Public bins can be **read and updated** from the browser (GET/PUT) without secrets.
- You'll use a **read** → **modify** → **PUT** cycle with an optimistic `rev` check (in your adapter) to prevent clobbering remote changes.
- No private keys are embedded in code.

Design decisions

- **Visibility:** Set the bin to **Public** in the dashboard so the adapter can GET/PUT from the browser.
- **Seed content:** Paste the **empty AppDoc** (which defines the sample schema).

```
{
    "version": 1,
    "rev": 1,
    "updatedAt": "1970-01-01T00:00:00.000Z",
    "users": [],
    "projects": [],
    "todos": [],
    "comments": [],
    "tags": []
}
```

Constraints & guardrails

- **Do not** attempt **POST** (create) or **DELETE** from the browser; those require keys. Use the dashboard to create/seed the bin.
- Use only **valid JSON** and avoid secrets—public data only.
- Keep the **document shape stable** (arrays exist) so CRUD code (`push`, `filter`) doesn't crash.

.

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** Create your own JSON BIN for Mock DB

Milestone 1: Create your own JSON BIN for Mock DB

Goal::

Create Create a PUBLIC BIN on JSON BIN that contains JSON representing the empty SCHEMA of your DB

Step 1:

Create a Free account: https://jsonbin.io/login

Step 2:

Go to Dashboard: https://jsonbin.io/dashboard

Step 3:

Click the "Bins" button in left menu bar

Step 4:

Click the "Create a Bin" button at top-right

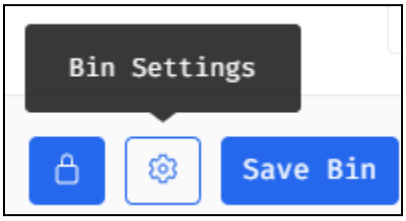
Step 4:

Enter valid JSON based on the Schema defined and populated by `seedDoc()`

```
{
  "version": 1,
  "rev": 1,
  "updatedAt": "1970-01-01T00:00:00.000Z",
  "users": [],
  "projects": [],
  "todos": [],
  "comments": [],
  "tags": []
}
```

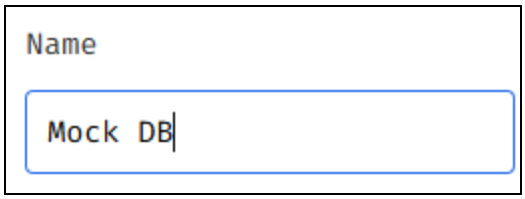
Step 5:

Click the "Bin Settings" button



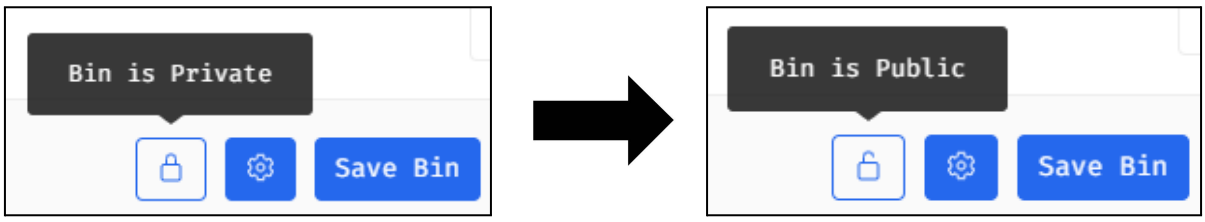
Step 6:

Enter a name for this BIN based on its usage



Step 7:

Click the visibility button for this bin to toggle it to PUBLIC BIN *(so the data is available without a key)*



Step 8:

Select the "Save Bin" button

Step 9:

JSON Bin is POSTed. Use the BIN ID to access it.

Milestone 1 — Testing

- **Check Public URL for your BIN:** https://api.jsonbin.io/v3/b/<BIN ID>

Goal 2: JsonBin Adapter

'Approach' → Plan phase

Goal

Implement a **cloud-backed, shared persistence** adapter that fulfills the same contract as other engines -- `load()`, `save(next)`, `reset()`, `snapshot()` -- so the **DB API stays identical** when you swap storage (Memory ↔ LocalStorage ↔ JsonBin).

Why (shared cloud persistence)
A **PUBLIC JSONBin** is a single document in the cloud. Any client with the same **BIN_ID** sees the **same data** (GET) and can **update it** (PUT). This enables multi-tab/multi-client demos without a server.

Design

- **Network I/O:** All operations are **async**; expect latency and handle non-2xx with clear errors.
- **Public bin only:** Client does **GET** `/b/<BIN_ID>/latest?meta=false` and **PUT** `/b/<BIN_ID>`. **POST/DELETE** (create/delete bins) require keys → do those in the **dashboard**, not the browser.
- **Optimistic concurrency:** In `save(next)`, **re-GET latest**, compare `remote.rev` to `next.rev`; if remote is newer, **throw** ("Remote is newer; reload/merge before saving."). Otherwise **stamp** `next` (bump `rev`, set `updatedAt`) and **PUT**.
- **Valid JSON only:** Body must be valid JSON—no JS, no comments. **Seed** the bin once with the **empty AppDoc** (arrays present, neutral ISO time), or provide a one-time `reset()` for instructors.
- **Security:** Treat the document as **public**—no secrets. Prefer **per-student/per-team bins**.

Class-based adapters (power & flexibility)

- **Instantiable by design:** `new JsonBinAdapter({ binId })` lets you create **multiple adapters** (e.g., different bins), or run **local + cloud** in parallel (with explicit sync buttons).
- Clear contract surface (`load/save/reset/snapshot`), private fields, and discoverable configuration—tooling-friendly for Java-first students.

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** JsonBin Adapter

Milestone 1: JsonBin Adapter

Goal:

Implement a **class-based JSONBin adapter** that reads/writes a PUBLIC bin via **GET /latest** and **PUT**, with an **optimistic `rev` check** to prevent stale overwrites. It must fulfill the shared contract: `load()`, `save(next)`, `reset()`, `snapshot()`, mirroring your LocalStorage adapter's structure and method names. This lets you swap LocalStorage ↔ JSONBin without changing any DB or CRUD code.

Step 1: Class definition (private fields)

.Define the adapter with private state for **bin id**, **root URL**, **stamping policy**, and **reset permission**.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
// scripts/adapters/jsonBinAdapter.js
// Persistent adapter (JSONBin) - Same contract: load(), save(next), reset(), snapshot()

import { seedDoc } from "../model.js";

export class JsonBinAdapter {
  #binId;
  #root;
  #stampOnSave;
  #allowReset;
}
```

Step 2: Constructor & method binding

Initialize configuration; bind public methods so passing them as callbacks won't lose `this`.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
constructor({binId, root = "https://api.jsonbin.io/v3", stampOnSave = true, allowReset = false} = {}) {
    if (!binId) throw new Error("JsonBinAdapter: 'binId' is required.");
    this.#binId = binId;
    this.#root = root.replace(/\/+$/, "");
    this.#stampOnSave = stampOnSave;
    this.#allowReset = allowReset;
    this.load = this.load.bind(this);
    this.save = this.save.bind(this);
    this.reset = this.reset.bind(this);
    this.snapshot = this.snapshot.bind(this);
}
```

Step 3: `#urlLatest()` — latest endpoint

Compute the GET latest URL once from config.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
#urlLatest() { return `${this.#root}/b/${this.#binId}/latest?meta=false`; }
```

Step 4: `#urlBin()` — write endpoint

Compute the **PUT** URL once from config.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
  #urlBin()    { return `${this.#root}/b/${this.#binId}`; }
```

Step 5: `#stamp(d)` — standardize rev/updatedAt

Bump `rev` and set `updatedAt` (UTC ISO) before persisting.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
  #stamp(d) {
    d.rev = (d.rev ?? 0) + 1;
    d.updatedAt = new Date().toISOString();
  }
```

Step 6: `#readLatest()` — GET latest from JSONBin

Fetch and return the current document; fail clearly on non-2xx.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```

  async #readLatest() {
    const res = await fetch(this.#urlLatest());
    if (!res.ok) throw new Error(`JSONBin read failed: ${res.status}`);
    return res.json();
  }
```

Step 7: `#write(next)` — PUT to JSONBin

Serialize and write the updated document to the bin; fail clearly on non-2xx.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
  async #write(next) {
    const res = await fetch(this.#urlBin(), {
      method: "PUT",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(next),
    });
    if (!res.ok) throw new Error(`JSONBin write failed: ${res.status}`);
  }
```

Step 8: `load()` — read latest

.Return the **latest** remote document (no mutation).

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
  // Load the latest document from the bin
  async load() {
    return await this.#readLatest();
  }
```

Step 9: `save(next)` — optimistic concurrency + write

Fetch **fresh remote**, compare `remote.rev` vs `next.rev`, and block on stale writes; stamp and PUT on success.

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
// Save with optimistic concurrency: compare remote.rev vs next.rev
async save(next) {
    // Fetch the freshest remote to avoid writing stale data
    const remote = await this.#readLatest();
    const rRev = remote?.rev ?? 0;
    const nRev = next?.rev ?? 0;

    if (rRev > nRev) {
      // Someone else wrote to the bin; caller should reload/merge
      throw new Error("Remote is newer; reload/merge before saving.");
    }

    if (this.#stampOnSave) this.#stamp(next);
    await this.#write(next);
}
```

Step 10: `reset()` — overwrite with a fresh seed (optional)

If enabled, write a **fresh** `seedDoc()` to the bin (useful for instructor/admin resets).

scripts/adapters/jsonBinAdapter.js → *JsonBinAdapter*

```
// Overwrite the bin with a fresh seed (disabled by default)
async reset() {
    if (!this.#allowReset) {
      throw new Error("JsonBinAdapter.reset(): disabled. Enable with { allowReset:true }.");
    }
    const fresh = seedDoc();
    if (this.#stampOnSave) this.#stamp(fresh);
    await this.#write(fresh);
}
```

Step 11: `snapshot()` — deep copy of current remote

Return a **deep copy** of the latest remote document for safe inspection (no mutation).

`scripts/adapters/jsonBinAdapter.js` → *JsonBinAdapter*

```
// Return a safe copy of the current remote doc (no mutation)
async snapshot() {
    const doc = await this.#readLatest();
    // structuredClone may not exist in some older engines; JSON dance is fine too
    return typeof structuredClone === "function" ? structuredClone(doc) :
JSON.parse(JSON.stringify(doc));
  }
}
```

Step 12: Convenience factory (back-compat usage)

Keep the familiar call style: `useAdapter(jsonBinAdapter(BIN_ID))`.

`scripts/adapters/jsonBinAdapter.js` → *JsonBinAdapter*

```
// Convenience factory to keep backward-compatible usage:  useAdapter(jsonBinAdapter(BIN#ID))
// You can still 'new JsonBinAdapter({ binId: BIN#ID, ...opts })' if preferred.
export function jsonBinAdapter(binId, opts = {}) {
    return new JsonBinAdapter({ binId, ...opts });
}
```

Milestone 1 — Testing

- **Tester:** The “Test — JSONBin section should pass with a green check.

'Approve' → Test phase

Test Instructions

1. Open **`/index.html`** ➜ click **Test — JSONBin**
    - Paste your **PUBLIC BIN_ID** in the tester's field, click **Boot**.
    - **Pass when:**
        - **Boot & Seed** is green (remote arrays exist; `rev/updatedAt` shown).
        - **CREATE / READ / UPDATE / DELETE** blocks pass.
        - **Advanced** blocks (Filter Objects, `find` projection/sort/paging, `updateOneOps` `$set/$addToSet/$pull`, `upsert`, `transact`) pass.
        - **Reboot/Consistency** check is green (after `boot()` the `rev` is same or newer).
2. Open **`/demos/jsonbin.html`** (interactive console).
    - Paste **BIN_ID**, click **Boot**.
    - Run **insertOne** → **getDoc**; refresh the page ➜ the record persists (cloud-backed).
    - Run **updateOneOps** (`$set/$addToSet/$pull`) and **transact**; confirm **`rev/updatedAt` advance** after each save.
    - (Optional) Open a **second tab** with the same BIN_ID, edit there, then try saving in the first tab ➜ you should see **"Remote is newer; reload/merge before saving."** Click **Boot** and retry.

Troubleshooting

- **403/404 or cannot boot:** Ensure the bin is **PUBLIC** and the **BIN_ID** is correct.
- **Validation errors:** The bin's content must be **valid JSON** (no JS) and match any attached **Schema Doc**.
- **Conflict error:** This is expected with concurrent edits; **Boot** (re-GET latest) and retry the save.
- **Network issues/latency:** JsonBin is **network-backed**; all calls are async -- retry after a moment.

Dev Cycle 5:
Sync & Multi-Adapter
(Local ⇄ Cloud)

What you're building (and why)

Add explicit **Sync Up / Sync Down** actions that copy the **entire AppDoc** between two adapters -- such as **LocalStorage** (fast, offline-ish, unlimited) and **JSONBin** (cloud, shared, rate-limited). This shows the **same DB API** can talk to multiple storage engines and that adapters are **instantiable and swappable**.

> **Shared cloud persistence:** a **Public JSONBin** bin is a **single shared document**. Anyone with the same **BIN_ID** sees the **same data** and can **update it**.
>
> **Local-first policy (quota-friendly):** Do routine edits in **LocalStorage**. **Publish** via **Sync Up** when needed (share/backup). **Pull** via **Sync Down** when switching devices or joining a team. This respects JSONBin's monthly rate limits by taking advantage of LocalStorage to aggregate cloud access into batches.

Objectives (end-of-cycle checklist)

By the end of this cycle you will be able to:

- Create two adapters (e.g., `new LocalStorageAdapter({ key })` and `new JsonBinAdapter({ binId })`) and **boot** either as the active engine.
- Implement **Sync Up** (Local → Cloud) and **Sync Down** (Cloud → Local) using the adapter **contract** only (`load`, `save`).
- Handle JSONBin **conflict errors** ("Remote is newer…") by **Booting** (re-GET) and **retrying** the operation.
- Verify sync in tests/demos: data moves correctly in **both directions**, and **rev/updatedAt** advance on the destination.

Table of Contents

Goal 1: Sync Data (Local ⇄ Cloud)

'Approach' → Plan phase

Goal

Add explicit, user-triggered **Sync Up (local → cloud)** and **Sync Down (cloud → local)** using the existing **adapter contract** (`load()`, `save(next)`), with clear conflict handling and **no secrets** in the client.

Design

- **Explicit actions:** Sync happens when the user clicks **Sync Up / Sync Down**, not automatically.
- **Whole-doc copy:** `source.load()` → deep clone → `dest.save(copy)`. No partial writes.
- **Conflicts:** Destination `save` uses the adapter's **optimistic `rev` check**. If it throws "Remote is newer," **Boot** (re-GET) the destination and retry.
- **Two patterns supported:**
    1. **Single DB context + sync helpers** (simplest): the DB uses **one active adapter**; sync helpers copy between **two adapter instances**.
    2. **Parallel DB contexts** (advanced): two independent DB instances each bound to a different adapter for side-by-side views.

'Apply' → Do phase

Overview of milestones

- **Milestone 1:** Sync functions (single DB context)

Milestone 1 — Sync functions (single DB context)

Goal:

Create tiny helpers that use the **adapter contract only** -- no changes to the DB or CRUD code.

Step 1: Implement syncUp

Create the `syncUp` function. This function reads the entire `AppDoc` from a source adapter (`readAdapter`), creates a safe deep copy, and then writes that copy to a destination adapter (`writeAdapter`). This function beautifully demonstrates the power of the adapter contract: it orchestrates persistence without knowing or caring what the underlying storage engines are.

`scripts/sync.js` → *syncUp*

```
// scripts/sync.js — explicit, user-triggered sync
export async function syncUp(readAdapter, writeAdapter) {
  const doc = await readAdapter.load();
  const copy = typeof structuredClone === "function"
    ? structuredClone(doc)
    : JSON.parse(JSON.stringify(doc));
  await writeAdapter.save(copy); // may stamp rev/updatedAt
}
```

Step 2: Implement syncDown

Create the `syncDown` function. The logic is identical to `syncUp`; only the source and destination adapters are swapped. This function is responsible for pulling data from a remote source (like JSONBin) and overwriting the local state.

`scripts/sync.js` → *syncDown*

```
export async function syncDown(readAdapter, writeAdapter) {
  const doc = await readAdapter.load();
  const copy = typeof structuredClone === "function"
    ? structuredClone(doc)
    : JSON.parse(JSON.stringify(doc));
  await writeAdapter.save(copy);
}
```

When to use:

- **Sync Up:** Local → JSONBin (publishing local changes, making a backup).
- **Sync Down:** JSONBin → Local (restoring from a backup, joining a team/public project).

Notes & cautions

- **Overwrite Risk:** `Sync Down` is a destructive action that replaces all local data. Real apps should prompt the user for confirmation first.
- **Latency & Quota:** JSONBin is network-backed and rate-limited. Syncs should be manual and infrequent.
- **Privacy:** Treat your JSONBin Public bin as completely public; never store secrets.
- **No Automatic Merge:** On a conflict, our strategy is to **Boot & Retry**. Advanced teams could implement custom merge policies as a stretch goal, but that is outside the scope of this lab.

Milestone 1 — Testing

- **Tester:** The “Test — Sync section should pass with a green check.

Conclusion

Final Comments

This lab was a multifaceted journey through modern web application architecture. By completing it, you have successfully met every one of the original learning objectives:

- **API Design:** You created a clean, reusable Data Layer by building a consistent CRUD and query API in `db.js` that serves as the single point of entry for all data operations.
- **Decoupled Architecture:** You implemented a true decoupled architecture using the swappable "adapter" pattern. This was proven in the final cycle, where your core DB logic could orchestrate a sync between `LocalStorage` and `JSONBin` without any modifications.
- **Local Persistence:** You learned to persist data locally by building the `LocalStorageAdapter`, which serializes the application's state to JSON, allowing it to survive page reloads.
- **Cloud Persistence & Asynchronous Operations:** You persisted data to a shared cloud source by building the `JsonBinAdapter`. In the process, you handled real-world asynchronous operations with `async/await` and `fetch`, and managed concurrency with an optimistic revision check.
- **Database Concepts:** You gained practical experience with core document database concepts by designing a schema and implementing a powerful, Mongo-like API for queries, projections, and updates.

Future Improvements (Beyond the Scope of this Lab)

The database module you've built is a powerful foundation. If you wanted to take it even further, you could explore these professional-grade features:

- **Schema Validation:** Implement a validation layer that ensures data written to the database (e.g., `insertOne`, `updateOne`) conforms to the expected schema.
- **Advanced Query Operators:** Add support for more complex Mongo-like operators, such as `$or` for compound queries or operators for updating nested objects.
- **Automatic Syncing:** Implement a "background sync" that automatically pushes local changes to the cloud on a timer or when the network is available.
- **Advanced Conflict Resolution:** Instead of just "reload and retry," you could implement a merge strategy that attempts to combine remote and local changes.
- **Performance Indexing:** Simulate a database index by creating a lookup map (e.g., a `Map` of `id` -> `document`) to speed up `findOne` operations, avoiding a full array scan.
- **More Robust Error Handling:** Define and throw custom error types for different failure scenarios (e.g., `ValidationError`, `ConflictError`).

Lab Submission

Push all of your lab files into your forked repository and close this Lab in your github issues. Mark it as complete.

Module 2 - Project:

You can use thie database module from this lab in your project for satisting both the requirements for data persistence and for HTTP request from a web service (if you use the cloud-based adapter)

Appendix A: Sample Payloads for Interactive Demos

Test Samples for the Interactive Demos

Use the following JSON payloads to test the functionality of your database in the interactive demo pages (`demos/*.html`). Simply copy and paste the content of a code block into the corresponding `textarea` on the page and click "Run".

`findMany` (Basic Read)

These examples use the `where` helper to build a predicate function.

```
// Find all todos assigned to Alice
{ "col": "todos", "where": { "field": "assignedTo", "op": "eq", "value": "u-alice" } }
```

```
// Find all completed todos
{ "col": "todos", "where": { "field": "done", "op": "eq", "value": true } }
```

```
// Find all users *except* Alice
{ "col": "users", "where": { "field": "name", "op": "neq", "value": "Alice" } }
```

```
// Find projects with "Site" in the name
{ "col": "projects", "where": { "field": "name", "op": "contains", "value": "Site" } }
```

```
// Find all documents in the 'tags' collection
{ "col": "tags", "where": null }
```

`insertOne` (Create)

Create new documents in different collections.

```
// Add a new user named "Charlie"

{ "col": "users", "data": { "name": "Charlie" } }
```

```
// Add a new high-priority todo with a due date

{ "col": "todos", "data": { "title": "Write report", "done": false, "assignedTo": "u-bob", "projectId": "p-1", "due": "2025-11-01" } }
```

```
// Add a new project

{ "col": "projects", "data": { "name": "Side Project", "ownerId": "u-bob", "tagIds": [] } }
```

```
// Add a todo with embedded subtasks

{ "col": "todos", "data": { "title": "Plan vacation", "done": false, "subtasks": [{"id": "s-v1", "title": "Book flights", "done": false}] } }
```

```
// Add a new "urgent" tag

{ "col": "tags", "data": { "name": "urgent" } }
```

`updateOne` (Basic Update)

Applies a shallow patch to an existing document.

```
// Mark the seed todo "t-1" as complete
{ "col": "todos", "id": "t-1", "patch": { "done": true } }
```

```
// Change the title of todo "t-1"
{ "col": "todos", "id": "t-1", "patch": { "title": "Repository has been set up" } }
```

```
// Reassign todo "t-1" to Bob
{ "col": "todos", "id": "t-1", "patch": { "assignedTo": "u-bob" } }
```

```
// Overwrite the tags array for project "p-1"
{ "col": "projects", "id": "p-1", "patch": { "tagIds": ["t-web", "t-setup"] } }
```

```
// Rename the user "u-bob" to "Robert"
{ "col": "users", "id": "u-bob", "patch": { "name": "Robert" } }
```

`/` (Advanced Read)

Query using Mongo-style filter objects.

```
// Find all incomplete todos assigned to Alice
{ "col": "todos", "filter": { "done": false, "assignedTo": "u-alice" } }
```

```
// Find user "Alice" or "Bob"
{ "col": "users", "filter": { "id": { "$in": ["u-alice", "u-bob"] } } }
```

```
// Find todos due on or after '2025-10-10'
{ "col": "todos", "filter": { "due": { "$gte": "2025-10-10" } } }
```

```
// Find the project that is NOT the "Course Site"
{ "col": "projects", "filter": { "name": { "$ne": "Course Site" } } }
```

```
// Find the first tag that contains "web" in its name
{ "col": "tags", "filter": { "name": { "$contains": "web" } } }
```

`find` (Combined Query: Filter, Sort, Page, Project)

The ultimate query tool for building lists.

```
// Get all incomplete todos, sorted by title (descending)
{ "col": "todos", "args": { "filter": { "done": false }, "sort": { "title": -1 } } }
```

```
// Get just the ID and title of all todos
{ "col": "todos", "args": { "fields": { "id": 1, "title": 1 } } }
```

```
// Get the second page of users (2 per page)
{ "col": "users", "args": { "limit": 2, "skip": 2 } }
```

```
// Get the titles of the first 2 incomplete todos, sorted by due date
{ "col": "todos", "args": { "filter": { "done": false }, "sort": { "due": 1 }, "limit": 2, "fields": { "title": 1 } } }
```

```
// Get all projects, but only show their name
{ "col": "projects", "args": { "fields": { "name": 1 } } }
```

`updateOneOps` (Advanced Update)

Use operators like `$set`, `$addToSet`, and `$pull`.

```
// On todo "t-1", mark it as done and add the "t-web" tag
{ "col": "todos", "id": "t-1", "ops": { "$set": { "done": true }, "$addToSet": { "tagIds": "t-web" } } }
```

```
// Try to add the "t-web" tag again (it shouldn't create a duplicate)
{ "col": "todos", "id": "t-1", "ops": { "$addToSet": { "tagIds": "t-web" } } }
```

```
// Remove the "t-web" tag from todo "t-1"
{ "col": "todos", "id": "t-1", "ops": { "$pull": { "tagIds": "t-web" } } }
```

```
// Change Bob's name back to "Bob" from "Robert"
{ "col": "users", "id": "u-bob", "ops": { "$set": { "name": "Bob" } } }
```

```
// Add multiple tags to a project at once
{ "col": "projects", "id": "p-1", "ops": { "$addToSet": { "tagIds": ["new-tag-1", "new-tag-2"] } } }
```

`upsertOne` (Create or Update)

Ensures a record matching the filter exists.

```
// First run: Creates a new "priority" tag
{ "col": "tags", "filter": { "name": "priority" }, "data": { "name": "priority" } }
```

```
// Second run: Finds the "priority" tag and updates it (no duplicate is created)
{ "col": "tags", "filter": { "name": "priority" }, "data": { "name": "priority" } }
```

```
// Find the user "Carol" and update her record, or create it if she doesn't exist
{ "col": "users", "filter": { "name": "Carol" }, "data": { "name": "Carol", "status": "active" } }
```

```
// Ensure the main project has the correct owner ID
{ "col": "projects", "filter": { "id": "p-1" }, "data": { "ownerId": "u-alice" } }
```

```
// Find a todo by title and mark it as done (will update if it exists)
{ "col": "todos", "filter": { "title": "Set up repo" }, "data": { "done": true } }
```

`transact` (Batch Operations)

The input is a string of JavaScript code to be executed inside

```
// Rename a project AND re-tag all of its todos in one save
const p = doc.projects.find(x => x.id === "p-1");
if (p) {
  p.name = "University Website";
  for (const t of doc.todos.filter(t => t.projectId === "p-1")) {
    t.tagIds = ["t-web", "t-setup", "t-urgent"];
  }
}
```

```
// Delete user "u-bob" and reassign their todos to "u-alice"
doc.todos.forEach(t => {
  if (t.assignedTo === "u-bob") {
    t.assignedTo = "u-alice";
  }
});
doc.users = doc.users.filter(u => u.id !== "u-bob");
```

```
// Create a new project and its first todo at the same time
const newProjectId = "p-new";
if (!doc.projects.find(p => p.id === newProjectId)) {
  doc.projects.push({ id: newProjectId, name: "New Initiative", ownerId: "u-alice", tagIds: [] });
  doc.todos.push({ id: "t-new-1", projectId: newProjectId, title: "Draft proposal", done: false });
}
```

```
// Mark two different todos as complete in a single operation
const todo1 = doc.todos.find(t => t.id === "t-1");
const todo2 = doc.todos.find(t => t.title.includes("syllabus")); // Find another todo
if (todo1) todo1.done = true;
if (todo2) todo2.done = true;
```

```
// Add a "archived" tag and apply it to all completed todos
if (!doc.tags.find(t => t.name === "archived")) {
  doc.tags.push({ id: "t-archived", name: "archived" });
}
doc.todos.forEach(t => {
  if (t.done) {
    const tagSet = new Set(t.tagIds || []);
    tagSet.add("t-archived");
    t.tagIds = Array.from(tagSet);
  }
});
```