Cloudflare D1 Course

Flavio Copes

Updated August 7, 2026

Contents

Build a relational Worker application with D1, SQLite, prepared statements, migrations, local and remote databases, transactions, backups, and production operations.

What you'll learn: Build, migrate, query, test, deploy, back up, and recover a small D1-backed application without mixing local and production data.

D1 foundations

Understand managed SQLite, databases, bindings, local state, and where D1 fits.

Understand where D1 fits

Use relational structure and SQLite semantics without treating D1 as KV, object storage, or a process-local file.

D1 is Cloudflare's managed relational database built on SQLite. You get tables, indexes, joins,

constraints, and transactions — real SQL, without provisioning a server, opening a network port, or managing a connection pool.

If you know SQLite, the query language is already familiar. What changes is the access model. A Worker talks to D1 through a **binding** rather than opening a local file:

```
const user = await env.DB.prepare(
  'select * from users where email = ?'
).bind(email).first()
```

There is no connection string and nothing to keep alive. `env.DB` is injected by the platform, configured once in `wrangler.jsonc`.

That distinction matters more than it looks. D1 is not a `.sqlite` file your process owns. It is a managed service with its own limits, its own transaction behavior, and a migration workflow. Habits from embedded SQLite — long-lived open handles, filesystem tricks, "just copy the file to back it up" — do not transfer.

Choose it for relational data

Choose D1 when rows, relationships, constraints, transactions, and SQL queries match the data. Users, notes, orders, settings: entities that reference each other and need queries like "the ten most recent notes by this user, with their tags."

```
select notes.title, group_concat(tags.name) as tags
from notes
join note_tags on note_tags.note_id = notes.id
join tags on tags.id = note_tags.tag_id
where notes.user_id = ?
group by notes.id
order by notes.created_at desc
limit 10;
```

That query is the argument for relational storage. Doing the same with key-value lookups means fetching everything and joining by hand in JavaScript.

Know what it is not

Each neighbor product keeps its own job. Choose R2 for file bodies — storing images as blobs in database rows wastes both products. Choose KV for read-heavy, eventually consistent values with no query needs, like feature flags. Choose Durable Objects when one entity needs serialized coordination, like a counter that must be exact under concurrency.

My rule of thumb: a key and a value, KV. Entities and relationships, D1. Files, R2. One hot coordinated thing, Durable Objects.

Now model a notes application with users, notes, and tags. Identify the primary keys, the relationships between the three tables, and one query that makes relational storage useful.

Create and bind local and remote databases

Keep development state separate from production while using one typed binding name in Worker code.

Create a D1 database with Wrangler, then add its identifier under a binding such as `DB` in

`wrangler.jsonc`. Regenerate Worker types after the configuration changes.

Local D1 commands and `wrangler dev` use local state by default. A remote command changes the deployed database. Make `--local` and `--remote` choices explicit in scripts and review output before applying data changes.

Create the practice database, apply one local command, and prove the remote database remains empty.

Keep the local and remote commands visibly different:

```
npx wrangler d1 create notes
npx wrangler d1 execute notes --local --command 'select 1'
npx wrangler d1 execute notes --remote --command 'select 1'
```

Add the returned binding to the Worker configuration. Run migrations locally first. Before every remote command, read the database name and the --remote flag aloud; this small habit prevents destructive experiments against the wrong database.

Check your understanding: d1 foundations

Check the key decisions, boundaries, commands, and failure cases from d1 foundations.

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

Take this interactive quiz online

Schema and migrations

Create a schema, version changes, apply migrations deliberately, and preserve compatibility.

Design a relational schema

Use keys, constraints, indexes, and timestamps to let the database protect important invariants.

A schema is more than column storage. It is the layer that protects your data when application code has a bug — and application code always eventually has a bug.

Primary keys identify rows. Foreign keys express relationships. `NOT NULL`, `UNIQUE`, and `CHECK` constraints reject invalid state even when a code path forgets to validate. Here is a notes schema that uses all of them:

```
create table users (
  id integer primary key autoincrement,
  email text not null unique,
  created_at integer not null
);

create table notes (
  id integer primary key autoincrement,
  user_id integer not null references users(id),
  title text not null check (length(title) <= 200),
  body text,
  created_at integer not null
```

```
);
```

Every rule here is one your route handlers would otherwise have to enforce in three different places. D1 enforces foreign key constraints, so an insert with a `user_id` that matches no user fails loudly:

```
D1_ERROR: FOREIGN KEY constraint failed
```

That error during development is the schema doing its job. The alternative is an orphaned note discovered months later.

Index what you actually query

Add indexes for real filter and ordering patterns, not every column:

```
create index idx_notes_user_created on notes (user_id, created_at desc);
```

This one serves "this user's notes, newest first," which is the query the application runs on every page load. An index speeds selected reads but costs storage and write work — each insert updates every index on the table. Indexing every column makes writes slow while helping no actual query.

Decide the boring things once

Keep timestamps in a documented format. I use integer Unix milliseconds and note that in the schema file; mixing ISO strings in some rows and integers in others makes sorting quietly wrong. SQLite's type system is permissive, so discipline has to come from you.

For multi-tenant data, define ownership from the start: which column marks the owning user or tenant, and every query filters by it. Retrofitting tenancy onto a shared table is far harder than including `user_id not null` on day one.

Now write the notes schema, then try to break it: insert a duplicate email, a note with a missing owner, and a title over 200 characters. Each attempt should fail with a constraint error before you build any routes.

Version the schema with migrations

Create ordered SQL migration files and roll out additive changes before code depends on them.

You do not create tables by hand in production. D1 migrations record schema changes as ordered SQL files, committed with the application, so a clean database can be reproduced from history alone.

Create one:

```
npx wrangler d1 migrations create my-app-db create_notes
#   Successfully created Migration '0001_create_notes.sql'
```

Wrangler writes a numbered file into the `migrations` directory. Put the SQL from the previous lesson inside it, then apply it locally:

```
npx wrangler d1 migrations apply my-app-db --local
#   Executing on local database my-app-db...
#
#   0001_create_notes.sql
#
```

When the code that uses the new schema is ready to deploy, apply the same files with `--remote`.

Wrangler tracks which migrations already ran in a bookkeeping table, so each file runs exactly once and always in order. `wrangler d1 migrations list my-app-db --local` shows what is still pending.

Two habits keep this system honest. Never edit a migration file that has already been applied anywhere — it will not re-run, so your file and the real schema silently diverge. Write a new migration instead. And apply locally before remotely, every time.

Roll out additive changes first

Deployments are not instant. For a short window, old and new Worker versions serve traffic against the same database. Do not assume old and new Worker versions disappear at the same instant.

So prefer additive rollout: add a nullable column or a new table, deploy code that works with and without it, backfill if needed, then enforce or remove old structure later:

```
-- 0002_add_archived_at.sql
alter table notes add column archived_at integer;
```

The old Worker ignores the new column and keeps working. A destructive change in one step — renaming a column the live code still reads — takes production down for exactly the length of your deploy window, which is the worst possible timing.

SQLite also restricts some `ALTER TABLE` operations, so bigger reshapes become create-new-table, copy, swap. Keep each migration small and review the SQL before it touches remote.

Now create the first migration for the notes schema, apply it locally, list the migration state, then delete your local database state and rebuild it from only the committed files.

Check your understanding: schema and migrations

Check the key decisions, boundaries, commands, and failure cases from schema and migrations.

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

Take this interactive quiz online

Queries and transactions

Use prepared statements, binding, batches, transactions, indexes, and query plans.

Prepare and bind every request value

Keep untrusted values separate from SQL structure and handle D1 result metadata explicitly.

Every value that arrives with a request is untrusted. The way you keep it harmless is structural: create statements with `env.DB.prepare()` and place request values in `bind()`.

```
const note = await env.DB.prepare(
  'select id, title, body from notes where id = ? and user_id = ?'
).bind(noteId, userId).first()
```

The SQL string contains only structure. The values travel separately, so a title like `'); drop table notes; --` is stored as those literal characters, never executed. Never build a SQL string by concatenating a title, ID, sort field, or tenant value — one template literal with `${userInput}`

inside is the entire SQL injection vulnerability class. This rule has no exceptions for values you believe are safe, because the next developer will not know why you believed it.

Pick the result shape on purpose

Each way of running a statement returns something different, and picking the right one removes a class of bugs:

```
const row  = await stmt.first()   // one object, or null
const all  = await stmt.all()     // { results, success, meta }
const raw  = await stmt.raw()     // arrays of values, no column names
const info = await stmt.run()     // meta for writes
```

`first()` returning `null` is your "not found" signal — handle it instead of reading properties off it. For writes, read the metadata:

```
const { meta } = await env.DB.prepare(
  'update notes set title = ? where id = ? and user_id = ?'
).bind(title, noteId, userId).run()

if (meta.changes === 0) {
  return new Response('Not found', { status: 404 })
}
```

An `UPDATE` matching zero rows is not an exception — it "succeeds" while changing nothing. A successful HTTP request should not hide a failed database operation, and `meta.changes` is how you notice. Without this check, an update to someone else's note returns 200 while doing nothing, and both the user and your logs believe it worked.

Identifiers cannot be bound

Bound parameters represent values, not table or column names. `order by ?` binds the string as a constant, so the sort silently does nothing. When the client picks a sort field, allowlist it:

```
const sortable = { date: 'created_at', title: 'title' }
const column = sortable[requestedSort] ?? 'created_at'
const rows = await env.DB.prepare(
  `select id, title from notes where user_id = ? order by ${column} desc`
).bind(userId).all()
```

The interpolated value comes from your own fixed map, never from the request.

Now implement note creation and lookup with bound parameters, then test quotes and injection-shaped text as ordinary data. The malicious string should come back from a `select` exactly as it was stored.

Group related writes and inspect plans

Use batch or transaction behavior for related changes and add indexes from measured query plans.

Related writes must succeed or fail together when partial state would be wrong. Creating a note and its audit record is one logical change; a crash between two separate statements leaves a note that officially never happened.

D1's tool for this is `batch()`. The statements run sequentially inside one transaction — if any

statement fails, the earlier ones roll back:

```
await env.DB.batch([
  env.DB.prepare(
    'insert into notes (user_id, title, created_at) values (?, ?, ?)'
  ).bind(userId, title, Date.now()),
  env.DB.prepare(
    'insert into audit_log (user_id, action, created_at) values (?, ?, ?)'
  ).bind(userId, 'note.created', Date.now()),
])
```

Note what `batch()` is not: an interactive transaction where you read, compute in JavaScript, then write. The statements are fixed up front. If a later statement needs a value a previous one produced, restructure — SQLite gives you tools like `last_insert_rowid()` within the batch.

Clients retry. A user double-clicks, a network hiccup replays a request, and your "one logical operation" runs twice. Make retries safe with an idempotency key: a unique token per logical operation, stored with a `UNIQUE` constraint, so the second attempt fails cleanly instead of duplicating data.

Read the plan before adding indexes

When a query feels slow, do not guess. Use `EXPLAIN QUERY PLAN` on important reads:

```
explain query plan
select * from notes where user_id = 42 order by created_at desc limit 10;
-- SCAN notes
```

`SCAN` means SQLite reads the whole table. Add the index that matches the filter and the ordering, then measure again:

```
create index idx_notes_user_created on notes (user_id, created_at desc);
-- after:
-- SEARCH notes USING INDEX idx_notes_user_created (user_id=?)
```

`SEARCH ... USING INDEX` is the confirmation. The pair of plans, before and after, is the evidence that the index earns its cost — because more indexes are not automatically faster. Each one slows every write, so an index that no query plan uses is pure overhead. Add an index when the plan and the workload justify it, and re-check plans when queries change.

Now create a note and its audit record as one batch, force the second statement to fail with a constraint violation, and verify neither row remains.

Check your understanding: queries and transactions

Check the key decisions, boundaries, commands, and failure cases from queries and transactions.

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

Take this interactive quiz online

Application patterns

Build CRUD, pagination, validation, authorization, idempotency, and optional ORM integration.

Build validated and authorized CRUD

Keep transport validation, tenant authorization, database constraints, and safe responses as separate layers.

Validate JSON before querying. Authenticate the caller, then include ownership or tenant boundaries in the SQL operation itself. Checking ownership in one query and updating in another can create gaps and duplicated logic.

Return 404 when a caller should not learn whether another tenant's row exists. Keep database errors out of public responses, but log safe identifiers and operation names for diagnosis.

Implement update and delete statements that include both note ID and owner ID in their `WHERE` clauses.

Bind values instead of building SQL strings:

```
const note = await env.DB
  .prepare('select id, title from notes where id = ? and user_id = ?')
  .bind(noteId, userId)
  .first()
```

The user ID must come from a verified identity, not a request field. Test a valid owner, another user, a missing note, and a malformed ID. Parameter binding protects the SQL boundary; the ownership predicate protects the authorization boundary.

Paginate and use an ORM deliberately

Choose stable pagination and treat Drizzle as a typed query tool rather than a replacement for SQL and migrations.

Any list that grows needs pagination, and the first rule is ordering: order every paginated query by a stable unique sequence. `order by created_at` alone is not stable when two notes share a timestamp — add the primary key as a tiebreaker, or page boundaries become random.

Offset pagination is easy but can drift as rows change:

```
select id, title, created_at from notes
where user_id = ?
order by created_at desc, id desc
limit 10 offset 20;
```

If a new note arrives while a user reads page 2, everything shifts and page 3 repeats an item they already saw. Offsets also get slower as they grow, because the database still walks past every skipped row.

Cursor pagination fits long or changing collections better. Instead of a page number, the client sends the sort values of the last item it saw:

```
select id, title, created_at from notes
where user_id = ? and (created_at, id) < (?, ?)
order by created_at desc, id desc
limit 10;
```

The query says "the next 10 older than this exact position." New rows cannot shift the window, and the index from the earlier lessons serves it at constant cost regardless of depth.

Drizzle is a tool, not a shield

Drizzle can define your schema in TypeScript and give you typed queries with autocomplete:

```
const rows = await db.select().from(notes)
  .where(eq(notes.userId, userId))
  .orderBy(desc(notes.createdAt), desc(notes.id))
  .limit(10)
```

That is genuinely useful — no raw string typos, column renames become compile errors. But the deployed database still follows SQL, indexes, constraints, and migration history. The ORM does not add an index you never created, and a Drizzle query producing `SCAN notes` is exactly as slow as the handwritten version.

Wiring also deserves attention: drizzle-kit generates SQL migration files into a directory, and Wrangler applies them, so verify the migration layout against current D1 and Wrangler configuration — the `migrations_dir` in `wrangler.jsonc` must point where drizzle-kit writes. Two tools each half-managing migrations is how schemas drift.

Now add cursor pagination to the notes list, then compare the SQL Drizzle emits with the handwritten version, and run both through `EXPLAIN QUERY PLAN` to confirm they use the same index.

Check your understanding: application patterns

Check the key decisions, boundaries, commands, and failure cases from application patterns.

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

Take this interactive quiz online

Test, back up, and operate D1

Test real bindings, inspect production, use time travel, observe failures, and recover.

Test the real schema and failure paths

Run integration tests against a clean D1 binding created from migrations and cover constraints, authorization, and retries.

Pure functions can use ordinary unit tests. Database behavior cannot — a mocked `env.DB` happily accepts queries your real schema would reject. Database behavior deserves tests in the Workers runtime with a D1 binding and the actual migration history.

Cloudflare's Vitest integration (`@cloudflare/vitest-pool-workers`) runs your tests inside the same runtime your Worker uses, with real bindings. Tests import the environment and apply your committed migrations before anything else runs:

```
import { env } from 'cloudflare:test'
import { beforeEach, expect, it } from 'vitest'

beforeEach(async () => {
  await env.DB.exec('delete from audit_log')
  await env.DB.exec('delete from notes')
})
```

Build fresh state per test. Each test seeds exactly the rows it needs and assumes nothing about leftovers. Avoid tests that pass only because a developer's old local database already contains the schema — that suite fails on every new machine and in CI, and nobody remembers why.

Test the paths that fail

The happy path is one test. The failure paths are where the schema and authorization work earn their keep:

```
it('rejects a note with a missing owner', async () => {
  const attempt = env.DB.prepare(
    'insert into notes (user_id, title, created_at) values (?, ?, ?)'
  ).bind(9999, 'orphan', Date.now()).run()

  await expect(attempt).rejects.toThrow(/FOREIGN KEY constraint failed/)
})
```

Cover the full list: duplicate values hitting a `UNIQUE` constraint, missing foreign rows, a transaction failure that must roll back its earlier statements, pagination boundaries (empty page, exact page size, past the end), unauthorized updates where `meta.changes` must be zero, and an idempotent retry that runs the same logical operation twice and asserts one row.

Each of these is a production incident you are choosing to have in a test instead.

Prove the suite owns its schema

The final check is reproducibility. Delete local test state and prove the suite recreates everything from committed migrations:

```
rm -rf .wrangler/state
npm test
#   all tests pass on a database built from migrations alone
```

If tests fail after this, they depended on state that only existed on your machine. Fix the setup, not the assertion.

Now write the failure-path suite for the notes schema, then run the wipe-and-rerun check above and make it part of how you work.

Back up, observe, and recover

Use current D1 recovery tools, safe query inspection, metrics, and a rehearsed restore plan before production needs them.

The moment to design recovery is before the destructive mistake, because during one you will be typing fast with bad information.

D1 gives you two recovery tools. **Time Travel** can restore a database to an earlier point in time, using its built-in change history:

```
npx wrangler d1 time-travel info my-app-db
#   Time Travel is currently active
#   Earliest restorable bookmark and timestamp shown here
```

The other is a plain SQL export you own:

```
npx wrangler d1 export my-app-db --remote --output=backup-2026-08-03.sql
```

The export is a file on your side — store it where your other backups live. D1's time-travel and backup capabilities evolve with the platform, and retention windows differ by plan, so check current retention and command documentation rather than copying an old number into the runbook. A runbook that says "we have 30 days" is only true until the plan or the product changes.

Restore into a separate target first

A restore is itself a destructive operation when pointed at the wrong place. Test recovery into a separate target before replacing good production data:

```
npx wrangler d1 create my-app-db-restore-test
npx wrangler d1 execute my-app-db-restore-test --remote --file=backup-2026-08-03.sql
npx wrangler d1 execute my-app-db-restore-test --remote \
  --command 'select count(*) from notes'
```

Now you know the backup actually restores, how long it takes, and what the row counts should look like — while production keeps running untouched. An untested backup is a hope, not a plan.

Observe without leaking

Log operation, duration, safe request ID, and failure class — never row contents or credentials. `note.create failed: UNIQUE constraint` is diagnosable; logging the note body is a data leak waiting for a log-access review. Monitor query latency, errors, storage size, and your position against current limits, and alert on the trend, not just the outage: a database growing toward a storage limit is a Tuesday problem or a 3am problem depending on when you notice.

Now run the full drill on a practice database: create disposable data, record a recovery point, make a destructive change like dropping a table, and restore or clone according to current official instructions. Time yourself — that number belongs in the runbook.

Check your understanding: test, back up, and operate d1

Check the key decisions, boundaries, commands, and failure cases from test, back up, and operate d1.

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

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/cloudflare-d1/.