Drizzle ORM Course

Flavio Copes

Updated August 3, 2026

Contents

Build a typed SQLite data layer with Drizzle ORM: schemas, migrations, CRUD, relations, transactions, tests, query plans, and production choices.

What you'll learn: Build, migrate, query, test, inspect, and prepare a complete Drizzle-powered notes data layer for deployment.

Drizzle foundations

Choose Drizzle deliberately, set up the project, open SQLite, and inspect the first query.

Choose Drizzle with open eyes

Understand what an ORM gives you, what Drizzle deliberately leaves visible, and when plain SQL may still be the better tool.

An ORM connects application objects and database rows. The dangerous promise is that you can stop thinking about SQL. You cannot.

Drizzle takes a smaller approach. You define the schema in TypeScript and build typed queries with familiar SQL operations. The database still owns constraints, transactions, indexes, and query plans. That is a feature: when a query becomes slow or wrong, you can reason about the SQL underneath it.

For this course we will build a small notes database with users and notes. SQLite keeps the setup local, while the Drizzle concepts transfer to PostgreSQL, MySQL, D1, and other supported drivers.

My advice is to use Drizzle when you want TypeScript help without hiding the database. Use plain SQL when the query is clearer that way. Drizzle includes an `sql` template for those cases, so this is not an all-or-nothing choice.

Write down one table and three queries from an application you know. For each query, name what TypeScript can verify and what only the database or a real test can verify.

Create the notes project

Set up a small TypeScript project with the current Drizzle 1.0 packages, Node SQLite, environment loading, and a repeatable command.

Let's get a real database running first. We will use Node's built-in SQLite driver, so there is no database server to install.

The current Drizzle 1.0 documentation may publish packages under the `rc` tag until the stable release. Use the versions shown in the official getting-started page and keep `drizzle-orm` and `drizzle-kit` on the same release line.

```
mkdir drizzle-notes
cd drizzle-notes
npm init -y
npm i drizzle-orm@rc dotenv
npm i -D drizzle-kit@rc tsx typescript @types/node
```

Create `src/db`, then add `DB_FILE_NAME=notes.sqlite` to `.env`. Keep the database file and `.env` out of Git. We will commit the schema and migrations instead, because those files describe how another developer can recreate the database.

If Drizzle 1.0 is stable when you take this course, the official page may omit `@rc`. Follow the current install command rather than forcing an old tag from a screenshot or tutorial.

Install the packages and save the exact versions from `npm ls drizzle-orm drizzle-kit`. Add a deliberate TypeScript error to your entry file and prove the project catches it before continuing.

Open SQLite with Drizzle

Create one database module that owns the SQLite connection and exports the Drizzle client used by the rest of the application.

The database client should have one obvious home. This keeps connection setup out of every query file.

Load the database filename from the environment and fail immediately when it is missing. Then pass the filename to the Node SQLite Drizzle driver. The returned `db` object builds queries and sends them through the underlying SQLite connection.

```
import 'dotenv/config'
import { drizzle } from 'drizzle-orm/node-sqlite'

if (!process.env.DB_FILE_NAME) {
  throw new Error('DB_FILE_NAME is required')
}

export const db = drizzle(process.env.DB_FILE_NAME)
```

A local file is convenient, but it is still state. Two tests that share it can affect each other. Later we will create isolated temporary databases for tests and keep production connection choices outside query code.

Do not place an untrusted request value into the database filename. The environment controls infrastructure; users control data passed to queries.

Run the module once with `DB_FILE_NAME` present and once without it. Save the created file and the startup failure, then confirm the file is ignored by Git.

Run and inspect the first query

Execute a small SQL expression through Drizzle and inspect generated queries instead of treating the ORM as an invisible layer.

Before creating tables, let's prove the complete path works. Drizzle can run a typed SQL template when the query builder is not the right tool.

Use the `sql` template to keep values separate from SQL text. It is not the same as string concatenation: interpolated values become bound parameters. Drizzle query builders also expose their generated SQL, which is useful when debugging a filter or reviewing a surprising query.

```
import { sql } from 'drizzle-orm'
import { db } from './db'

const result = await db.execute(sql`select sqlite_version() as version`)
console.log(result)
```

A successful result proves the TypeScript entry point, Drizzle driver, SQLite library, environment, and database file can work together. When this fails, inspect those layers in that order instead of rewriting schema code that has not run yet.

Raw SQL is useful for database-specific features and diagnostics. Keep it narrow, bind every value, and return to the query builder when its types make the application clearer.

Print the SQLite version and one bound value containing a quote. Save the output and the generated SQL representation, then prove the quoted value remains data rather than changing the statement.

Check your understanding: drizzle foundations

Check what Drizzle does, how it stays connected to SQL, and how a TypeScript project opens and inspects SQLite.

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

Declare tables and constraints, configure Drizzle Kit, and apply reviewed migrations.

Define users and notes

Describe two related SQLite tables in TypeScript while keeping their database names, required values, and foreign key visible.

A Drizzle schema is executable TypeScript that describes database tables. It is not a replacement for database design.

Create `src/db/schema.ts`. Users have an email and display name. Notes belong to a user through `authorId`. The foreign key protects the relationship even when another script writes directly to SQLite.

```
import { int, sqliteTable, text } from 'drizzle-orm/sqlite-core'

export const users = sqliteTable('users', {
  id: int().primaryKey({ autoIncrement: true }),
  email: text().notNull().unique(),
  name: text().notNull(),
})

export const notes = sqliteTable('notes', {
  id: int().primaryKey({ autoIncrement: true }),
  authorId: int('author_id').notNull().references(() => users.id),
  title: text().notNull(),
  body: text().notNull().default(''),
})
```

TypeScript now knows which fields exist and whether they can be null. SQLite still enforces `NOT NULL`, uniqueness, and the foreign key at runtime. We want both layers because either one can be bypassed independently.

A relation used for convenient fetching does not create a database foreign key. The `.references()` call does. We will add Drizzle relations later, after the database rules are correct.

Add the two tables, then intentionally remove `authorId` from a note value and inspect the TypeScript error. Later, after migrating, attempt an invalid author ID and save the database error too.

Put rules in the schema

Use defaults, uniqueness, checks, foreign keys, and indexes for rules that must survive every application write path.

Application validation gives friendly errors. Database constraints protect the data when another path forgets that validation.

Keep required fields `notNull()`, make email unique, and choose delete behavior deliberately. Add an index only for a query pattern you can name. For our app, listing one user's notes by creation time is that pattern.

```
import { index, int, sqliteTable, text } from 'drizzle-orm/sqlite-core'
import { sql } from 'drizzle-orm'

export const notes = sqliteTable('notes', {
  id: int().primaryKey({ autoIncrement: true }),
  authorId: int('author_id').notNull().references(() => users.id, { onDelete: 'cascade' }),
  title: text().notNull(),
  body: text().notNull().default(''),
  createdAt: text('created_at').notNull().default(sql`CURRENT_TIMESTAMP`),
}, table => [
  index('notes_author_created_idx').on(table.authorId, table.createdAt),
])
```

Cascade delete means deleting a user deletes their notes. That may fit a disposable notes app and be unacceptable for invoices or audit records. The ORM cannot make this product decision for you.

Every index speeds some reads and adds work to writes. Start from a real filter and sort order, then inspect the plan after data exists.

Write one sentence justifying every constraint and the composite index. Remove the index temporarily and compare the query plan for one author's newest notes after inserting enough rows to make the difference visible.

Configure Drizzle Kit

Point Drizzle Kit at the schema, migration directory, SQLite dialect, and database without leaking credentials into version control.

Drizzle ORM runs application queries. Drizzle Kit compares schemas and manages migration files. They solve different jobs.

Create `drizzle.config.ts` at the project root. Load the same environment value as the application, then declare the schema path, output folder, dialect, and database URL. Commit the config but not the local database or secrets.

```
import 'dotenv/config'
import { defineConfig } from 'drizzle-kit'

export default defineConfig({
  schema: './src/db/schema.ts',
  out: './drizzle',
  dialect: 'sqlite',
```

```
  dbCredentials: {
    url: process.env.DB_FILE_NAME!,
  },
})
```

The non-null assertion keeps the example small, but configuration should still fail clearly when the variable is absent. You can validate before exporting the config when several environments share the project.

For PostgreSQL the dialect and credentials change. The application schema also uses `pgTable` and PostgreSQL column builders. Drizzle makes the workflow familiar, not identical across databases.

Run a Drizzle Kit command with the correct environment, then rename the schema path and database variable one at a time. Save both failures so you can distinguish a missing schema from a missing connection.

Generate, review, and apply migrations

Turn a TypeScript schema change into reviewed SQL, apply it once, and understand when direct schema push is the wrong workflow.

A migration is a database change with history. Treat generated SQL as a draft you must understand before it touches important data.

Run `generate` after changing the schema. Drizzle Kit compares its snapshots and writes SQL. Read that SQL, commit it with the schema change, then run `migrate`. Drizzle records applied migrations so the same file is not applied twice.

```
npx drizzle-kit generate --name=initial_schema
# Read the generated SQL in ./drizzle
npx drizzle-kit migrate
```

`drizzle-kit push` skips versioned SQL files and updates the database from the schema difference. That is fast for disposable prototypes. For a shared or production database, reviewed migrations give you evidence, coordination, and a repeatable deployment path.

A migration that adds a required column to a populated table may fail or invent unsafe data. Plan the data transition in steps instead of accepting a generated statement blindly.

Generate and apply the initial migration, then add an optional `archivedAt` column. Review the SQL before applying it and prove a second migrate run makes no duplicate change.

Check your understanding: schema and migrations

Check schema declarations, database constraints, indexes, Drizzle Kit configuration, and the generate-review-migrate workflow.

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

Take this interactive quiz online

Typed CRUD

Insert, select, filter, update, and delete rows through narrow typed queries.

Insert users and notes

Insert typed values, let the database own generated columns, and use returned rows instead of making a fragile second lookup.

An insert should contain the values the caller owns. IDs, timestamps, and defaults can stay with the database.

Drizzle derives the accepted insert shape from the table. Required values are required by TypeScript, while generated and defaulted columns may be omitted. SQLite supports `returning()`, so we can receive the row created by this statement.

```
const [user] = await db
  .insert(users)
  .values({
    email: 'flavio@flaviocopes.com',
    name: 'Flavio',
  })
  .returning()
```

A unique email conflict is an expected database outcome. Translate it into a useful application error without exposing raw SQL or assuming every database error means the same thing.

Never spread an untrusted request object directly into `.values()`. Build the allowed object field by field after runtime validation. TypeScript checks your code, not the JSON sent by a browser.

Insert one user and one note for that user, then save the returned rows. Attempt a duplicate email and an unknown author ID, and record which database rule rejects each write.

Select, filter, order, and limit

Build a bounded query for one user's newest notes and inspect its result type and generated SQL.

A useful collection query needs a filter, stable order, and limit. Returning every row is not a harmless default.

Use operators such as `eq()` instead of writing condition strings. Select only the columns the caller needs, order newest first, and add the ID as a tie-breaker when timestamps can match.

```
import { and, desc, eq, isNull } from 'drizzle-orm'

const rows = await db
  .select({ id: notes.id, title: notes.title })
  .from(notes)
  .where(and(eq(notes.authorId, userId), isNull(notes.archivedAt)))
  .orderBy(desc(notes.createdAt), desc(notes.id))
  .limit(20)
```

Use `isNull()` for null checks rather than copying `= NULL` SQL. SQL null has three-valued logic, so ordinary equality is not the right comparison.

The selected object shape becomes the result type. This is useful for privacy too: if a list does not need note bodies, do not select and move them through the application.

Seed 25 notes with repeated timestamps, then request the first two pages using a bounded limit. Save the generated SQL and prove the order stays deterministic across repeated runs.

Update one owned note

Use a narrow WHERE clause and returned rows so an update proves both ownership and whether a matching record existed.

The most expensive missing line in an update is often `where()`. Without it, every row can change.

For user-owned data, include both the note ID and trusted author ID in the condition. Do not load the row, check ownership, and update later when one conditional statement can keep that decision atomic.

```
const [updated] = await db
  .update(notes)
  .set({ title: 'A better title' })
  .where(and(eq(notes.id, noteId), eq(notes.authorId, userId)))
  .returning()
```

When `updated` is undefined, the note may be missing or belong to another user. Returning the same public result for both cases avoids revealing another user's object existence.

Drizzle prevents many column mistakes, but it cannot know whether `userId` came from a verified session or an untrusted request body. Authorization still belongs to the application boundary.

Update a note as its owner, another signed-in user, and an unknown ID. Save the returned rows and final database state, then add a test that would fail if the ownership condition disappeared.

Delete with a deliberate boundary

Delete only the intended row, consider soft deletion honestly, and verify the effect instead of assuming a successful query changed data.

Delete is another mutation where a missing condition can destroy the complete table. Keep the boundary visible.

Use the same ID and owner condition as the update. `returning()` tells us whether a row was deleted. A soft-delete timestamp can help recovery, but it adds a filter to every read and does not replace backups or retention rules.

```
const [deleted] = await db
  .delete(notes)
  .where(and(eq(notes.id, noteId), eq(notes.authorId, userId)))
  .returning({ id: notes.id })
```

Some teams forbid update or delete without `where()` through linting or a wrapper. That guard is useful, but it cannot tell whether the condition is correct. A condition such as `id > 0` still matches every normal row.

Choose hard delete, archive, or retention based on the product. Do not add soft deletion automatically and then forget that private data remains stored.

Delete one owned note and try the same operation as another user. Prove the second attempt changes nothing, then explain whether this product needs hard deletion, archiving, or time-limited recovery.

Check your understanding: typed crud

Check inserts, selects, filters, ordering, updates, deletes, returning rows, and the safety boundary around every mutation.

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

Take this interactive quiz online

Relations and transactions

Connect users and notes, compare joins, and keep related writes atomic.

Separate foreign keys and relations

Understand the different jobs of database foreign keys and Drizzle relations before using convenient nested queries.

Foreign keys and Drizzle relations describe the same connection at different layers. They are not interchangeable.

The foreign key on `notes.authorId` prevents invalid data in SQLite. A Drizzle relation tells the query API how to fetch an author with notes or a note with its author. Relations do not create constraints or migrations.

This distinction matters when an import script, admin tool, or second service writes outside Drizzle. The database constraint still protects the relationship. An application-only relation does not.

Keep the foreign key even when the relational query API works without it. Only omit database enforcement when you have a concrete distributed-data reason and another integrity plan.

Draw the users-to-notes relationship twice: once as a database constraint and once as a fetch rule. Remove each layer in a disposable branch and record which invalid write or convenient query stops working.

Define one-to-many relations

Use the current Drizzle 1.0 defineRelations API to connect users and notes in one type-safe relation map.

Drizzle 1.0 collects relations in one `defineRelations()` call. Older tutorials use a different API, so check the installed major version before copying them.

Pass the schema object first. Then define both directions: a user has many notes, and each note has one author. The `from` and `to` columns make the connection explicit.

```
import { defineRelations } from 'drizzle-orm'
import * as schema from './schema'

export const relations = defineRelations(schema, r => ({
  users: {
    notes: r.many.notes({
      from: r.users.id,
      to: r.notes.authorId,
```

```
    }),
  },
  notes: {
    author: r.one.users({
      from: r.notes.authorId,
      to: r.users.id,
      optional: false,
    }),
  },
}))
```

`optional: false` is a type promise that an author always exists. It fits our non-null foreign key. Do not use it to hide nullable or inconsistent data, because the runtime row still decides what exists.

Pass the relations to the Drizzle client according to the current driver documentation. Then the `db.query` API can include nested records with typed results.

Add the relation map and intentionally swap one `from` column. Use the TypeScript error or a failing relational test to explain why each direction matters, then restore the correct mapping.

Choose a join or relational query

Fetch notes with authors using both explicit joins and the relational API, then choose from the result shape you need.

A join and a relational query can answer the same question. The clearer result shape should drive the choice.

An explicit join keeps selected columns and SQL structure visible. The relational API is convenient when the application wants nested objects. Neither approach removes the need to limit rows, filter ownership, or inspect performance.

```
const rows = await db
  .select({
    title: notes.title,
    authorName: users.name,
  })
  .from(notes)
  .innerJoin(users, eq(notes.authorId, users.id))
```

For an export with two columns, the explicit join is easy to read. For a user page containing a user object and a notes array, `db.query.users.findFirst({ with: { notes: true } })` may match the application shape better.

Do not fetch every column and discard most of them later. The query should express the data boundary, especially when rows contain private or large fields.

Implement the same “newest notes with author” screen using a join and a relational query. Compare generated SQL, result shape, selected columns, and query plan before choosing one.

Make multiple writes atomic

Use a transaction so creating a user and their first note either completes together or leaves no partial state.

Two successful statements are not one successful operation. The process can fail between them.

A transaction groups statements into one atomic unit. If the callback throws, Drizzle asks the driver to roll back. This is the right boundary when the product considers both writes one operation.

```
const result = await db.transaction(async tx => {
  const [user] = await tx
    .insert(users)
    .values({ email, name })
    .returning()

  const [note] = await tx
    .insert(notes)
    .values({ authorId: user.id, title: 'Welcome' })
    .returning()

  return { user, note }
})
```

Keep network calls outside database transactions. Waiting for email or an API holds locks and can fail in ways the database cannot roll back. Commit database state, then hand external work to a reliable follow-up mechanism.

Transaction behavior differs by database and driver. D1 batches, SQLite transactions, and PostgreSQL transactions do not have identical capabilities. Read the driver documentation before assuming one example transfers unchanged.

Run the transaction successfully, then force the note insert to violate a constraint. Prove neither row remains after the failure and save the exact database state, not only the thrown error.

Check your understanding: relations and transactions

Check foreign keys, Drizzle 1.0 relations, joins, relational queries, transactions, and atomic failure behavior.

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

Take this interactive quiz online

Test, inspect, and ship

Infer useful types, test real databases, inspect plans, and change drivers deliberately.

Infer select and insert types

Derive database row and insert types from the schema without confusing those types with validated request input or public responses.

The schema already knows its row shapes. Rewriting the same TypeScript interfaces by hand creates another source of truth.

Use a table's inferred select and insert types for internal database boundaries. Select types include returned columns. Insert types reflect generated and defaulted fields. They still do not validate

JSON at runtime.

```
export type Note = typeof notes.$inferSelect
export type NewNote = typeof notes.$inferInsert
```

A public API response may deliberately omit `authorId` or add a computed URL. Define that contract separately instead of returning database rows by accident.

Likewise, an HTTP create-note input should not include generated IDs or trusted ownership fields. Validate the request schema, then map it into `NewNote` with the server-owned author ID.

Create `Note`, `NewNote`, `CreateNoteInput`, and `PublicNote` types for the project. Add one private database field and prove it does not silently appear in the public mapping.

Test real queries in isolation

Run migrations and repository operations against a fresh temporary SQLite database so tests catch real schema and query mistakes.

A mocked Drizzle client can confirm a method call. It cannot prove the migration, SQL, constraints, and driver agree.

Create a temporary database for each test file or case, apply the real migrations, run the repository operation, and remove the file afterward. Keep tests independent so order never changes the result.

Test the failures the schema promises: duplicate email, missing author, ownership-bound update, and transaction rollback. A green insert test alone does not verify the important boundaries.

Using an in-memory database is fast, but a file-backed temporary database may better match configuration such as WAL mode and connection behavior. Choose deliberately and document the difference.

Write one integration test that migrates a fresh database, inserts a user and note, rejects an invalid author, and removes its file. Run two copies concurrently and prove they share no state.

Inspect SQL, plans, and data

Use generated SQL, SQLite query plans, measured data, and Drizzle Studio to diagnose queries before guessing at indexes.

Type-safe SQL can still be slow SQL. Types prove shapes, not performance.

Start with the generated statement and bound parameters. Run `EXPLAIN QUERY PLAN` against representative data. Drizzle Studio can help inspect rows and schema during development, but it does not replace a saved plan or a repeatable performance test.

A query over twenty rows will look fast with or without a useful index. Seed enough realistic data to expose the access pattern, then compare the plan and timing before and after the index.

Be careful with Studio against shared databases. A convenient data editor still has the authority of its credentials. Use the narrowest environment and access level that solves the debugging task.

Capture the generated SQL and query plan for one user's newest notes. Remove the composite index, compare the evidence, restore it, and explain the plan change in plain language.

Change drivers and ship safely

Separate portable Drizzle habits from database-specific schema, transaction, connection, and migration decisions before deploying.

Drizzle supports many databases, but changing the driver is not the same as changing an import.

PostgreSQL uses `pgTable`, its own column builders, a network connection or pool, and PostgreSQL migration SQL. Cloudflare D1 uses SQLite schema builders and a D1 binding, but its batching and deployment workflow differ from local Node SQLite.

Keep repository functions small and pass the database client where tests need isolation. Keep connection creation near the runtime boundary. This makes a driver change visible without pretending the databases behave identically.

For production, run reviewed migrations as a controlled release step, back up important data, monitor errors and latency, and rehearse recovery. Never run destructive schema push automatically against a database you cannot recreate.

Choose Node SQLite, PostgreSQL, or D1 for one real application and justify concurrency, hosting, backup, and migration needs. Sketch the changed client and schema imports, then write a deploy and rollback checklist.

Check your understanding: test, inspect, and ship

Check inferred types, isolated database tests, generated SQL, query plans, indexes, driver changes, migration releases, and production boundaries.

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/drizzle/.