Drizzle ORM

Version 1.0.0-rc.5 (stable: 0.45.4) · verified 2026-11-02
Typed SQL for TypeScript — schema, drizzle-kit migrations, queries and joins, relational queries, transactions, prepared statements and D1 notes.

Contents

Mental model 2

Versions and setup 2

Schema is the source of types 3

Migrations with drizzle-kit 5

Select, insert, update, delete 6

Joins and partial selects 7

Relational queries 8

Transactions and prepared statements 10

SQLite and Cloudflare D1 11

Drizzle vs Prisma 12

Further reading 13

Mental model

Drizzle is typed SQL, not an object mapper. The query builder mirrors SQL clause for clause, so if you can write the SQL you can write the Drizzle, and the table definitions in your schema file are the single source of both the database DDL and the TypeScript types. There is no identity map, no lazy loading, and no hidden query — every `await` sends exactly the statement the builder describes.

Versions and setup

Channel	Version	Install
`latest` (stable)	0.45.4	`drizzle-orm`
`rc`	1.0.0-rc.5	`drizzle-orm@rc`
Kit (`rc`)	1.0.0-rc.5	`drizzle-kit@rc`

Drizzle 1.0 is **not stable** on 2026-11-02: npm's `latest` tag is still 0.45.4, while the official docs are written for the 1.0 release candidate. The schema, query-builder and transaction samples compile against both; relational queries differ, and that section says how. Later samples import `db` from this `db.ts` and the tables from the schema below.

```
// db.ts — npm i drizzle-orm@rc pg
import { drizzle } from "drizzle-orm/node-postgres";


export const db = drizzle({
```

```
  connection: process.env.DATABASE_URL!,
});
```

Schema is the source of types

Builder	Produces
`pgTable("name", { … }, (t) => […])`	Table, columns, indexes
`.notNull()`, `.default()`	Non-null; optional on insert
`.generatedAlwaysAsIdentity()`	Identity column
`.references(() => t.id)`	Foreign key constraint
`index()`, `uniqueIndex()`, `check()`	Third-argument extras
`$inferSelect`, `$inferInsert`	Row and insert types

Each column builder adds both a DDL clause and a type refinement. A nullable column becomes `T | null` on select; a column with a default or identity becomes optional on insert. Export the inferred types instead of hand-writing interfaces that drift.

```
// schema.ts
import {
  pgTable, integer, text, timestamp,
} from "drizzle-orm/pg-core";

export const users = pgTable("users", {
  id: integer().primaryKey()
    .generatedAlwaysAsIdentity(),
  email: text().notNull().unique(),
```

```
  name: text(),
  createdAt: timestamp("created_at", {
    withTimezone: true,
  }).notNull().defaultNow(),
});
export type User = typeof users.$inferSelect;
```

Indexes, composite keys and checks go in the third argument, a function of the columns that returns an array.

```
// schema.ts, below users
import { sql } from "drizzle-orm";
import {
  pgTable, integer, text, index, check,
} from "drizzle-orm/pg-core";
export const posts = pgTable("posts", {
  id: integer().primaryKey()
    .generatedAlwaysAsIdentity(),
  authorId: integer("author_id").notNull()
    .references(() => users.id),
  title: text().notNull(),
}, (t) => [
  index("posts_author_idx").on(t.authorId),
  check("title_ok", sql`${t.title} <> ''`),
]);
```

> **Gotcha:** `timestamp()` without `withTimezone: true` is Postgres `timestamp`, not `timestamptz` — the offset is silently dropped. Drizzle defaults to the SQL default, not the safe one.

Migrations with drizzle-kit

Command	Does
`drizzle-kit generate`	Diff schema, write SQL migration
`drizzle-kit migrate`	Apply pending migration files
`drizzle-kit push`	Diff and apply directly, no files
`drizzle-kit pull`	Introspect a DB into a schema
`drizzle-kit check`	Detect conflicting migrations

`generate` compares your schema against its last snapshot and writes a reviewable SQL file; `migrate` applies the files not yet recorded in the database. `push` introspects the live database and applies the diff in one step — Drizzle's "code first" flow for prototyping. 1.0 changed the migration folder layout; run `drizzle-kit up` once when upgrading.

```
// drizzle.config.ts
import { defineConfig } from "drizzle-kit";

export default defineConfig({
  dialect: "postgresql",
  schema: "./src/schema.ts",
  out: "./drizzle",
```

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

> **Warning:** `push` leaves no migration file behind, so production and staging can drift with no record of what changed. Use `generate` plus `migrate` for any database you cannot recreate.

Select, insert, update, delete

Builder	SQL
`db.select().from(t).where(…)`	`SELECT … WHERE`
`db.insert(t).values(…).returning()`	`INSERT … RETURNING`
`.onConflictDoUpdate({ target, set })`	`ON CONFLICT … DO UPDATE`
`db.update(t).set(…).where(…)`	`UPDATE … SET`
`db.delete(t).where(…)`	`DELETE`

Operators such as `eq`, `and`, `inArray` and `isNull` come from `drizzle-orm`; the `sql` template is the escape hatch, and interpolated values in it are always sent as bind parameters, never concatenated. Results are plain arrays of plain objects.

```
import { eq, sql } from "drizzle-orm";
import { users } from "./schema";
import { db } from "./db";
```

```ts
const [u] = await db.insert(users)
  .values({ email: "a@x.io", name: "Ann" })
  .onConflictDoUpdate({
    target: users.email,
    set: { name: sql`excluded.name` },
  })
  .returning();

await db.update(users)
  .set({ name: "Ann B." })
  .where(eq(users.id, u.id));
```

> **Gotcha:** `db.update(users).set({...})` without `.where()` is a valid `UPDATE` of every row, and `db.delete(users)` empties the table. The builder mirrors SQL, including its sharp edges.

Joins and partial selects

Builder	Result row shape
`.innerJoin(t, on)`	`{ a: A; t: T }`
`.leftJoin(t, on)`	`{ a: A; t: T | null }`
`select({ id: a.id, … })`	Exactly the named fields

A join returns one flat row per SQL row, keyed by table name — Drizzle does not nest children under parents for you. A left-joined table is typed

nullable as a whole, which forces you to handle the no-match row that SQL actually returns.

```
import { eq } from "drizzle-orm";
import { users, posts } from "./schema";
import { db } from "./db";

const rows = await db
  .select({
    email: users.email,
    title: posts.title,
  })
  .from(users)
  .leftJoin(posts, eq(posts.authorId, users.id));
// { email: string; title: string | null }[]
```

Relational queries

API	Version
`defineRelations(schema, r => …)`	1.0 (RQB v2)
`relations(table, ({ one, many }) => …)`	0.45 (RQB v1)
`db.query.t.findMany({ with })`	Both
`where: { id: 1 }` object filters	1.0

The relational query API (RQB) returns nested objects — a user with an array of posts — from one SQL statement. It needs relations passed to

`drizzle()`. In 1.0 they are declared once for the whole schema with `defineRelations`, and `where`/`orderBy` take plain objects.

```
import { defineRelations } from "drizzle-orm";
import { users, posts } from "./schema";

const relations = defineRelations(
  { users, posts },
  (r) => ({
    users: { posts: r.many.posts() },
    posts: {
      author: r.one.users({
        from: r.posts.authorId,
        to: r.users.id,
      }),
    },
  }),
);
```

Query it with `drizzle({ connection, relations })`, then `db.query.users.findMany({ with: { posts: true } })`.

> **Gotcha:** relations are application-level only. They "do not create foreign keys implicitly" — add `.references()` on the column if you want the database to enforce the link.

Transactions and prepared statements

API	Does
`db.transaction(async (tx) => …)`	`BEGIN … COMMIT`
`tx.rollback()`	Throws, rolls back
`tx.transaction(…)`	Savepoint
`.prepare("name")`	Server-side prepared statement
`sql.placeholder("id")`	Parameter of a prepared query

Use `tx`, not `db`, inside the callback — a query on `db` runs on a different connection, outside the transaction. The callback's return value becomes the transaction's result; any throw rolls back. Prepared statements build the SQL string once and let the driver reuse it.

```
import { eq, sql } from "drizzle-orm";
import { users } from "./schema";
import { db } from "./db";

const byId = db.select().from(users)
  .where(eq(users.id, sql.placeholder("id")))
  .prepare("user_by_id");

const name = await db.transaction(async (tx) => {
  await tx.update(users).set({ name: "Bo" })
    .where(eq(users.id, 1));
  return "Bo";
```

```
}, { isolationLevel: "serializable" });


const [u] = await byId.execute({ id: 1 });
```

SQLite and Cloudflare D1

Postgres	SQLite / D1
`drizzle-orm/pg-core`, `pgTable`	`drizzle-orm/sqlite-core`, `sqliteTable`
`dialect: "postgresql"`	`dialect: "sqlite"`
`db.transaction(…)`	`db.batch([…])` on D1
Native `boolean`, `timestamptz`	`integer({ mode: "boolean" })` etc.

The query builder is the same; the column builders and drivers are not. On D1 you pass the Worker binding to `drizzle()` from `drizzle-orm/d1`. A D1 batch is one SQL transaction: if any statement fails, the whole sequence is rolled back. drizzle-kit reaches D1 over HTTP with `driver: "d1-http"`.

```
import { drizzle } from "drizzle-orm/d1";
import {
  sqliteTable, integer, text,
} from "drizzle-orm/sqlite-core";


const notes = sqliteTable("notes", {
  id: integer().primaryKey(),
```

```
  body: text().notNull(),
  done: integer({ mode: "boolean" }).default(false),
});

type Env = { DB: D1Database };
const run = (env: Env) =>
  drizzle(env.DB).select().from(notes).all();
```

Drizzle vs Prisma

	Drizzle	Prisma
Schema	TypeScript tables	`.prisma` file
Types	Inferred, no codegen	Generated by a CLI step
Queries	SQL-shaped builder + RQB	Object API

Pick Drizzle when your team thinks in SQL and wants the types without a generate step; pick Prisma when you want a higher-level object API and its own migration tooling. Prisma is mid-transition too — Prisma ORM 8 was a release candidate on writing day. Either way, read the SQL your ORM emits; in Drizzle, `toSQL()` shows it without running the query.

```
import { eq } from "drizzle-orm";
import { users } from "./schema";
import { db } from "./db";
```

```ts
const q = db.select().from(users)
  .where(eq(users.id, 1));
console.log(q.toSQL());
// { sql: 'select … where "users"."id" = $1',
//   params: [1] }
```

Further reading

- Drizzle ORM — Get started with PostgreSQL
- Drizzle ORM — Relational queries (RQB v2)
- Drizzle ORM — Upgrading to v1
- Drizzle ORM — Relations v1 to v2 migration
- Drizzle ORM — Cloudflare D1