

@mostajs/orm

Plug & Play ORM to Drive 13 Databases at Once

npm v2.5.3 downloads 3.1k/month License AGPL 3.0 dialects 13 types TypeScript minzipped size 6.6 KiB

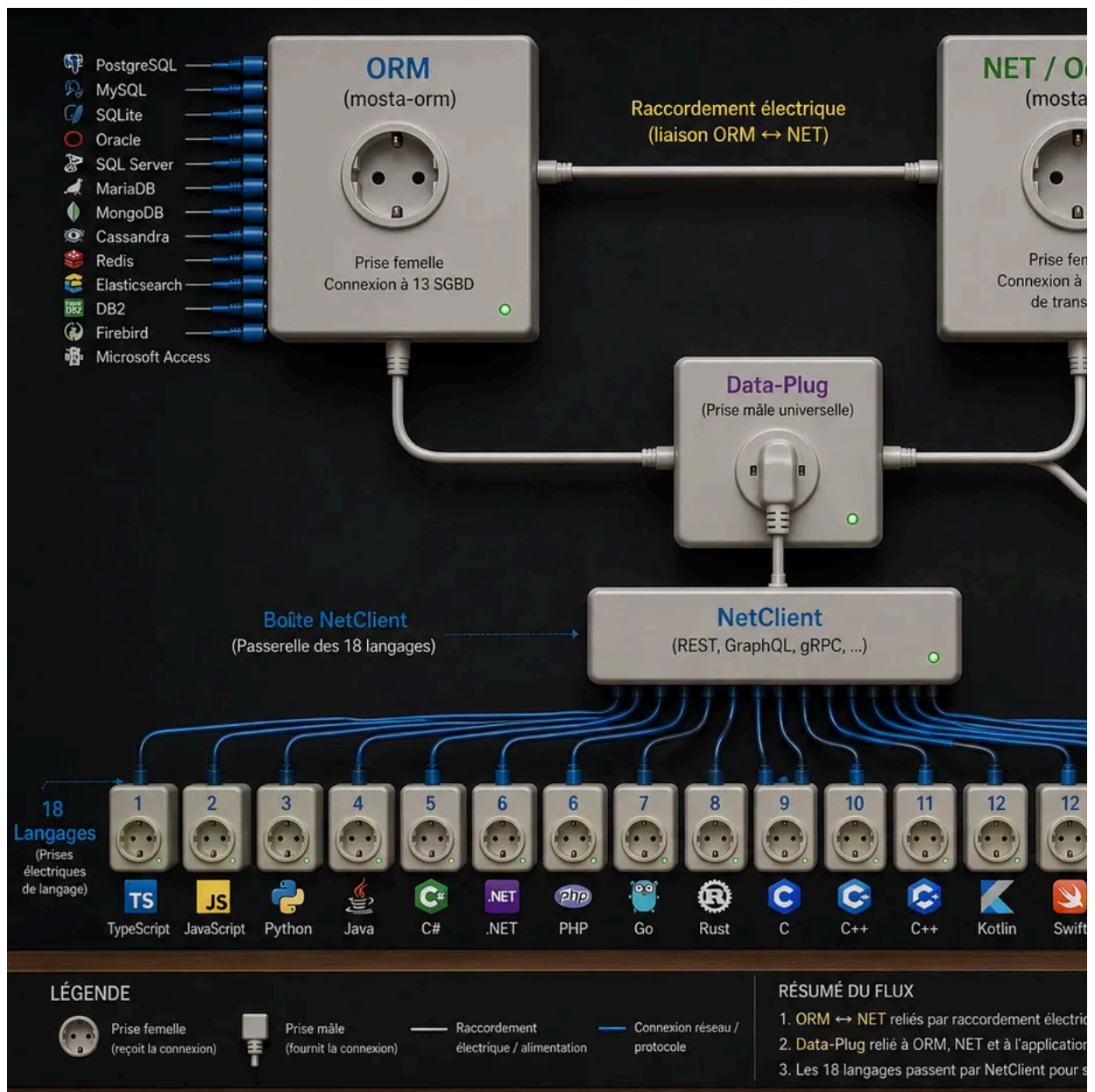
Hibernate-inspired multi-dialect ORM for Node.js & TypeScript — **one API, 13 databases, zero lock-in, bundler-friendly.**

npm · <https://www.npmjs.com/package/@mostajs/orm> **GitHub** · <https://github.com/apolocine/mosta-orm> **Docs** · (coming soon) **Product Hunt** · (launch link to be added)

Runnable samples · [@mostajs/orm-samples](#) — copy-paste install per feature, covering 100% of this package's public API.

```
npx @mostajs/orm-samples list # browse available samples
npx @mostajs/orm-samples scaffold 01-quickstart-sqlite ~/my-app
cd ~/my-app && ./01-quickstart-sqlite.sh # runnable in 30 seconds
```

The @mostajs stack at a glance



The @mostajs stack as electrical sockets: @mostajs/orm plugs into 13 databases, @mostajs/net (OctoNet) exposes them over 11 transports, Data-Plug bridges them, and NetClient fans out to 18 language clients consumed by any external app.

One socket board, any plug. @mostajs/orm connects to **13 databases**; @mostajs/net (OctoNet) re-exposes the same entities over **11 transports** (REST, WebSocket, gRPC, MQTT...); and NetClient fans out to **18 language clients** — so any external app talks to your data, in any runtime.

Why @mostajs/orm ?

- 🎯 **One API, 13 dialects.** Switch from PostgreSQL to MongoDB to SQLite without rewriting a single repository call.
- 🍷 **Zero lock-in.** Native drivers, no proprietary query DSL — your SQL/NoSQL stays portable.
- 🐘 **Hibernate / JPA semantics.** @OneToMany, cascade types, SAVEPOINT, schema strategies (validate / update / create / create-drop) — concepts battle-tested for 25 years, ported to TypeScript.
- 📦 **Drop-in Prisma replacement.** @mostajs/orm-bridge lets you keep your Prisma code while running on any of 13 databases.
- 🔄 **Cross-dialect replication built-in.** @mostajs/replicator — CDC + master/slave + failover across SQL ↔ MongoDB.
- 🛠️ **Bundler-friendly.** Tree-shakable ESM, no eval, works with esbuild / Vite / Next.js / Bun out of the box.
- 🏠 **Multi-app DB cohabitation (v2.3.0+).** DB_TABLE_PREFIX à la Hibernate physical_naming_strategy — let two apps share one Oracle/MSSQL/HANA DB user without colliding on users / roles / permissions.

60-second demo

```
npm install @mostajs/orm better-sqlite3
```

```
import { getDialect } from '@mostajs/orm'
import { UserSchema } from './schemas/user.schema'

const db = await getDialect({ dialect: 'sqlite', uri: ':memory:' }, [UserSchema])
const userRepo = db.repo<typeof UserSchema>('User')

await userRepo.create({ email: 'alice@example.com', name: 'Alice' })
const alice = await userRepo.findOne({ email: 'alice@example.com' })
```

Want PostgreSQL instead ? Change one line :

```
const db = await getDialect({ dialect: 'postgres', uri: process.env.DATABASE_URL }, [UserSchema])
```

That's it. Same repo.create(), same repo.findOne(), same TypeScript types — different dialect.

Need it to **run in the browser**, Bolt.new / StackBlitz or Cloudflare Workers ? Use the WASM dialect — **boots in the browser / Bolt.new / Cloudflare Workers with no native binary** :

```
npm install @mostajs/orm sql.js
```

```
// `sqljs` = SQLite compiled to WebAssembly. No `.node` addon → works where
// better-sqlite3 can't load (browser, WebContainer, edge). Same API, same SQL.
const db = await getDialect({ dialect: 'sqljs', uri: ':memory:' }, [UserSchema])
```

Starters — open in your browser

All boot in your browser with no native binary, on the first try, via the sqljs (SQLite WASM) dialect.

🚀 **Startup MVP starter** — a full foundation (landing + auth + CRUD dashboard, "rename & go") to launch a SaaS. Auth is powered by @mostajs/auth-lite (email/password + sessions, no native addon):

⚡ OPEN IN [STACKBLITZ](#) ⚡ OPEN IN [BOLT.NEW](#)  OPEN IN [CODESandbox](#)

📊 **Survey starter** — a customer-satisfaction survey (Next.js 15): landing, 5-question form, thank-you page & admin dashboard with bar charts. Rename & go:

⚡ OPEN IN [STACKBLITZ](#) ⚡ OPEN IN [BOLT.NEW](#)  OPEN IN [CODESandbox](#)

Or a minimal **blog** (Users · Posts · Comments — relations, soft-delete, seeded data) across four frameworks:

Starter	StackBlitz	Bolt.new	CodeSandbox
Next.js 15 (App Router)	Open StackBlitz	Open Bolt.new	Open CodeSandbox
Express	Open StackBlitz	Open Bolt.new	Open CodeSandbox
Fastify	Open StackBlitz	Open Bolt.new	Open CodeSandbox
Hono (Node / edge)	Open StackBlitz	Open Bolt.new	Open CodeSandbox

Tested E2E — the 6 starters × 3 gen-AI WebContainers (June 2026, [@mostajs/orm@2.5.3](#)):

Starter	StackBlitz	Bolt.new	CodeSandbox
nextjs-mostajs-orm-starter (blog)	✓	⚠ prod	✓
express-mostajs-orm-starter (blog)	✓	✓	✓
fastify-mostajs-orm-starter (blog)	✓	✓	✓
hono-mostajs-orm-starter (blog)	✓	✓	✓
mostajs-saas-starter (SaaS, auth)	✓	⚠ prod	✓
mostajs-survey-starter (survey)	✓	⚠ prod	✓
Total	6/6	3/6 (+3 Next en prod)	6/6

- ✓ = boot + full E2E scenario in `npm run dev`.
- ⚠ prod = Bolt's runtime **doesn't run Next 15 in dev** (`Error: Method not implemented.` from Bolt's bundle, not the starter) → run the **Next** starters in **prod** there (`npm run build && npm run start`). The **server** starters (Express/Fastify/Hono) run on Bolt as-is.
- **StackBlitz** = reference (best for debugging — terminal shows logs). **CodeSandbox** serves a cross-site iframe → the SaaS auth uses `crossSiteCookie` (`sameSite=none; secure`) to keep sessions.

Working from an **AI dev tool** (Cursor, Cline, Claude...)? Generate schemas, lint them (24 rules) and produce migrations via the MCP server [@mostajs/orm-mcp](#) — hosted at <https://orm-mcp.amia.fr/mcp>.

End-to-end, in one folder — sample `18-mcp-to-running-app` walks the whole chain: [@mostajs/orm-mcp](#) generates the `EntitySchema`, [@mostajs/orm](#) applies them, and an e-commerce app (users/products/orders) runs on `sqljs` (SQLite WASM, zero native binary). Ships a scripted HTML proof report.

How it compares

	@mostajs/orm	Prisma	Drizzle	TypeORM
SQL dialects	9 (PG, MySQL, MariaDB, SQLite, MSSQL, Oracle, DB2, HANA, Cockroach...)	5	5	8
NoSQL dialects	MongoDB native	✗	✗	✗
Same API across SQL & NoSQL	✓	✗	✗	✗
Browser / WebContainer / edge	✓ (WASM <code>sqljs</code> — zero native binary)	⚠ (Accelerate, paid)	⚠ (driver)	✗
Cross-dialect replication	✓ (via @mostajs/replicator)	✗	✗	✗
Schema-as-code (no DSL)	✓ TypeScript objects	DSL <code>.prisma</code>	TS objects	Decorators
Code generation step	✗ (zero codegen)	✓ required	✗	✗
Drop-in Prisma replacement	✓ (via @mostajs/orm-bridge)	—	✗	✗
Migration from Prisma	✓ (automated CLI)	—	manual	manual
Hibernate / JPA semantics	✓	✗	✗	partial
License	AGPL-3.0 (+ commercial)	Apache-2.0	Apache-2.0	MIT

Numbers as of v1.13.1 — see `@mostajs/orm-cli` for the automated Prisma → @mostajs migration tool.

The relation lookup problem (and how `@mostajs/orm@2.0` solves it)

The problem nobody talks about

Every ORM that auto-populates relations on read creates the **same silent bug class** :

```
// 'reg.project' is a M2O relation to Project
const reg = await regRepo.findById(regId)

// Eager mode (Hibernate ≤ JPA 2.x default, @mostajs/orm < 2.0) :
reg.project === project.id // false - object vs string, ALWAYS
reg.project.id === project.id // true if eager - explodes if lazy

// Lazy mode (Prisma, Drizzle, TypeORM 0.3+, SQLAlchemy default) :
reg.project === project.id // true if same id - but reg.project.name throws
```

3 surfaces affected : 1. **Direct comparison** `entity.relation === id` — JS has no operator overloading. 2. **Property access** `entity.relation.someField` — assumes populated. 3. **Reuse in lookup** `findById(entity.relation)` — assumes string id.

The default eager/lazy choice forces every consumer to be defensive everywhere.

How each ORM addresses it

ORM	Default fetch	Polymorphic key lookup	Helper to normalize id	Documented as problem
Hibernate	EAGER (historic JPA, anti-pattern)	<code>EntityManager.find(Class, key)</code> accepts <code>EmbeddedId</code>	overridable <code>equals(Object o)</code> in Java	Yes — Vlad Mihalcea has 50+ blog posts on it
Prisma	LAZY (opt-in <code>include</code>)	<code>findUnique({ where })</code> accepts any unique field	✗ none — <code>where: { id }</code> required	Partially — <code>findUnique</code> is the only escape
Drizzle	LAZY (opt-in <code>with</code>)	<code>query.x.findFirst({ where: eq(...) })</code> — verbose	✗ none	No — relation queries are explicit only
TypeORM ≥ 0.3	LAZY (opt-in <code>relations: ['rel']</code>)	<code>findOneBy({ id })</code> only	✗ none	No
MikroORM	LAZY (opt-in <code>populate</code>)	<code>findOne({ id })</code> only	<code>Reference<T></code> proxy (use <code>.id</code> accessor)	Yes — Reference helper documented
SQLAlchemy	LAZY (opt-in <code>joinedload</code>)	<code>session.get(Cls, ident)</code> accepts tuple for composite	✗ none in Python (no <code>==</code> overloading either)	Yes — community workarounds
<code>@mostajs/orm@2.1</code>	LAZY (opt-in <code>fetch: 'eager'</code>)	<code>findById()</code> polymorphe (string, <code>{id}</code> , natural key single or composite)	<code>extractRelId(value)</code> helper	Yes — explicitly documented + <code>ORMConceptValidator R019/R020/R021</code> (livrés en 2.1.0)

The `@mostajs/orm@2.0` 3-layer solution

Layer 1 — `lazy` by default

Aligns with Prisma / Drizzle / TypeORM 0.3+ / SQLAlchemy / MikroORM. Eliminates 90% of accidental N+1 queries and type confusion.

```
// Default - no surprise :
const reg = await regRepo.findById(regId)
typeof reg.project // 'string' - the FK id
```

Layer 2 — Polymorphic `findById` with schema introspection

Inspired by JPA `EntityManager.find(Class, Object)` and Prisma `findUnique({ where })`.

```
// String PK (legacy) – unchanged :
await projRepo.findById('abc-123')

// Object with `id` – natural for code that passes populated entities :
await projRepo.findById({ id: 'abc-123' })

// Natural key – schema unique index detected automatically :
await projRepo.findById({ slug: 'my-project' })

// Composite natural key :
await membershipRepo.findById({ tenantId: 't1', slug: 'admin' })
```

If the input is an object that matches neither `id` nor a unique index, `OrmIntrospectionError` is thrown with the available fields and candidate unique indexes listed — actionable error message.

Layer 3 — `extractRelId()` helper for direct comparisons

JS has no operator overloading. `obj === string` is always false. So we provide the explicit normalizer :

```
import { extractRelId } from '@mostajs/orm'

// Safe under both lazy AND eager :
if (extractRelId(reg.project) === project.id) {
  // ...
}
```

`extractRelId(value)` returns : - `value` itself if string/number stringified - `value.id` stringified if object with `id` - `''` for null / undefined / object without `id`

What `@mostajs/orm` does that no other JS/TS ORM does

Capability	@mostajs/orm 2.0	Others
Lazy default (<i>state of the art</i>)	✓	Prisma, Drizzle, TypeORM 0.3+, MikroORM, SQLAlchemy ✓
Opt-in eager via schema flag	✓ <code>fetch: 'eager'</code>	TypeORM via <code>eager: true</code> . Others : per-query only.
<code>findById</code> accepts string OR <code>{id}</code> OR natural key OR composite	✓	None (Prisma <code>findUnique</code> is closest but doesn't accept <code>{id}</code> object pass-through)
Public <code>extractRelId</code> helper for <code>===</code> comparisons	✓	None (MikroORM Reference is closest but requires API discipline)
Validator rule auto-detects the trap (<i>R021-DIRECT-RELATION-COMPARISON</i> , livré 2.1.0)	✓ via ORMConceptValidator	None
Auto-fix the trap (<i>injects <code>extractRelId</code> import + wraps comparison</i>)	⚠ V2 — diff suggéré fourni, application manuelle ou via plugin IDE	None

Benchmark — same code in 3 ORMs (*eager opt-in scenario*)

Scenario : compare a registration's project FK to a known project id, then re-fetch the project. Eager loading enabled for performance reasons.

```
// Prisma (eager not native – must include + spread) :
const reg = await prisma.registration.findUnique({
  where: { id }, include: { project: true }
})
if (reg.projectId === project.id) { /* hand-extract from FK field */ }
// re-fetch project : await prisma.project.findUnique({ where: { id: reg.projectId } })

// TypeORM 0.3+ (eager: true in @ManyToOne) :
const reg = await regRepo.findOne({ where: { id }, relations: ['project'] })
if (reg.project.id === project.id) { /* explicit .id needed */ }
// re-fetch project : redundant – reg.project is already loaded

// @mostajs/orm 2.0 (fetch: 'eager' in schema) :
const reg = await regRepo.findById(id)
if (extractRelId(reg.project) === project.id) { /* safe under any default */ }
// re-fetch project : await projRepo.findById(reg.project) ← introspection, works
```

Both `findById(reg.project)` and `extractRelId(reg.project)` work identically whether the relation is lazy (*string*) or eager (*object*). Consumer code does NOT need to know the fetch mode.

→ `@mostajs/orm` is the only JS/TS ORM where you can flip lazy ↔ eager via a schema flag without rewriting consumer code.

Star · Sponsor · Contribute

If `@mostajs/orm` saves you days of glue code, please :

- ★ **Star** the repo — visibility helps me keep maintaining it.
 - ❤️ **Sponsor** development → github.com/sponsors/apolocene
 - 🐛 Report issues / submit PRs — every contribution counts.
 - ✉️ Commercial license & support : drmdh@msn.com
-

Databases

SQLite · PostgreSQL · MySQL · MariaDB · MongoDB · Oracle · SQL Server · CockroachDB · DB2 · SAP HANA · HSQLDB · Spanner · Sybase

+ **WASM runtimes** — two zero-binary dialects run in WebAssembly, so the same ORM **boots in the browser / Bolt.new / Cloudflare Workers with no native binary**:

- `sqljs` — SQLite in WASM (via `sql.js`). In-memory in the browser; file-backed on Node.
- `pglite` — PostgreSQL in WASM (via `@electric-sql/pglite`). In-memory, `idb://` for durable in-browser storage (IndexedDB), or a directory on Node.

These are **not** new databases — they're zero-binary WASM runtimes of the SQLite/Postgres engines already listed, for environments where `better-sqlite3` / `pg` can't load.

Local-first, offline & embedded

Because the WASM build needs **no native binary and no server**, the same typed API runs on constrained targets that can't compile or ship a native SQLite addon:

- **Local-first / offline / PWA apps with no backend** — a note-taking editor, an offline field tool, an in-browser playground.
- **Embedded & IoT** — surveillance and agriculture drones, access-control gates and turnstiles, smartphones used as access badges, and any device with a JS/WASM runtime.

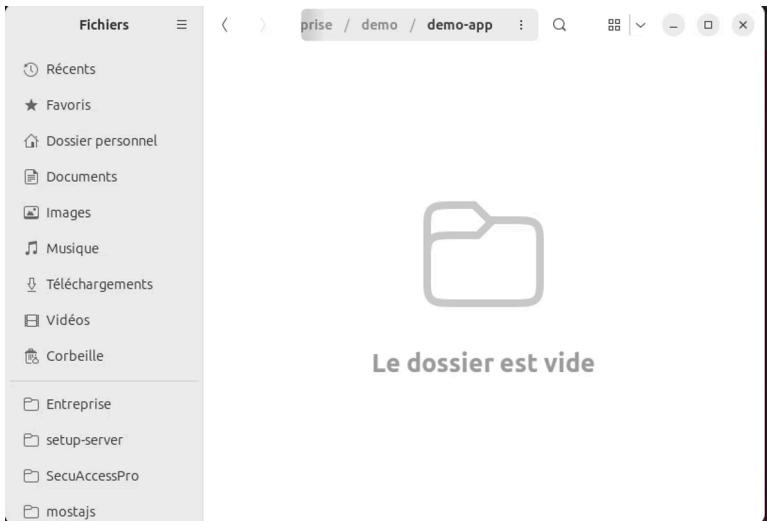
Marketing angle: *the only multi-dialect ORM that runs in the browser — and persists there* (`pglite` with `uri: 'idb://...'`, IndexedDB) — today. A real differentiator vs Prisma/Drizzle, claimed honestly: in-browser execution + durable storage ship now via `pglite`; for `sqljs`, browser persistence (IndexedDB/OPFS) is a planned opt-in (today it is in-memory in the browser, file-backed on Node).

Use with AI dev tools

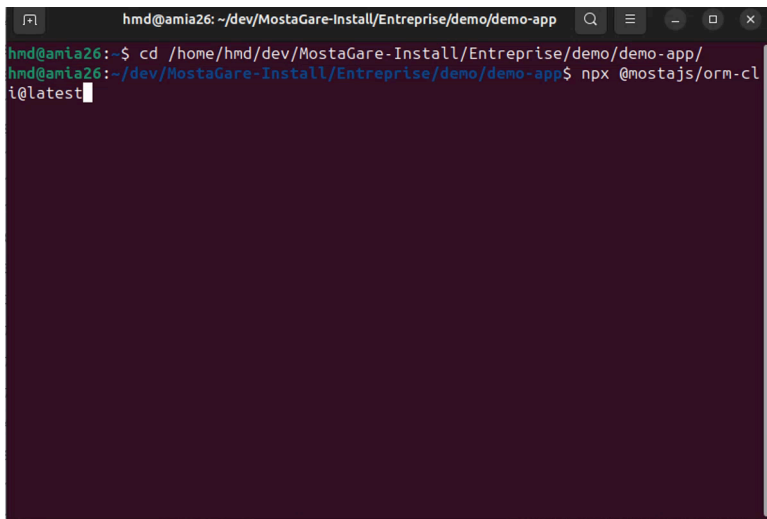
- **Bolt.new · StackBlitz · CodeSandbox** — open a `sqljs` starter by URL (bolt.new/github.com/apolocene/nextjs-mostajs-orm-starter); it boots with no native binary.
 - **Cursor · Cline · Claude Code** — first-class schema/migration/validation tooling via the `@mostajs/orm-mcp` server — **live** in the official MCP Registry, hosted at <https://orm-mcp.amia.fr/mcp>. See the end-to-end sample `18-mcp-to-running-app`. Prefer plain codegen? Point them at `llms.txt`.
-

Demos

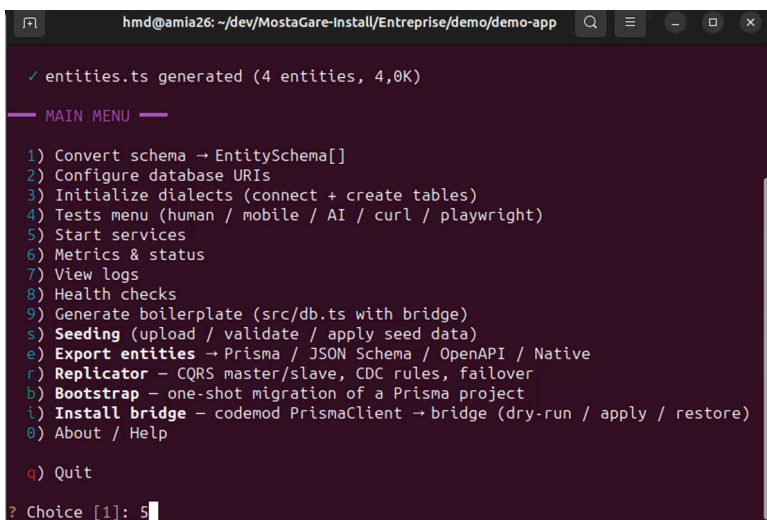
1. Initialize the app



2. Initialize the database

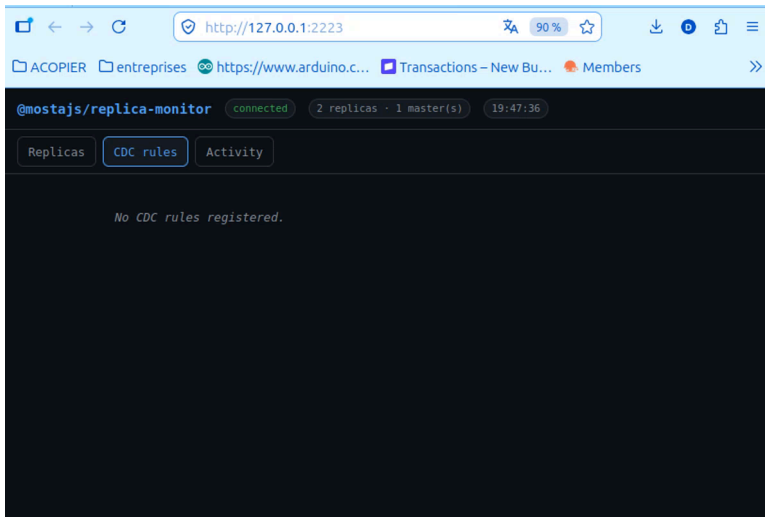


3. Configure the app

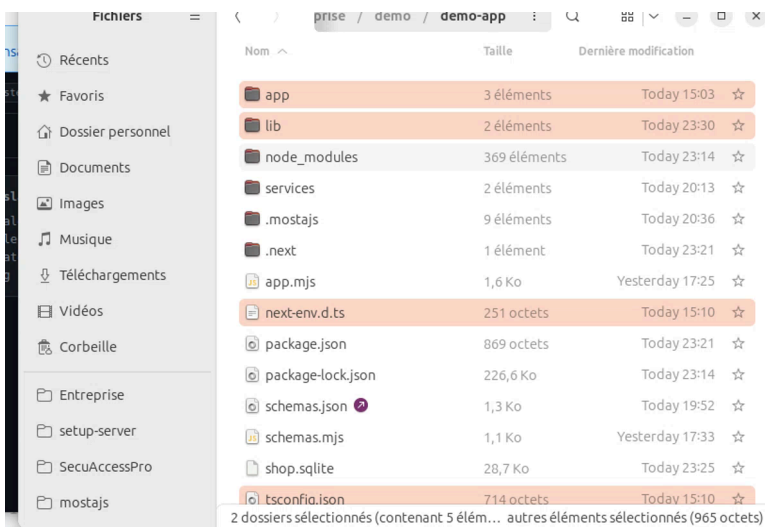


4. Setup replication

5. Cross-dialect CDC rules & live sync



6. Frontend CRUD app



7. Prisma project (before migration)

Dossier personnel / DEMO-00 / ecom-saas-prisma			Q	☰	▼	⊞	×
Nom	Taille	Dernière modification					
prisma	2 éléments	Today 17:45	☆				
src	2 éléments	Today 17:14	☆				
next.config.mjs	41 octets	Today 17:14	☆				
package.json	474 octets	Today 17:14	☆				
README.md	5,6 Ko	Today 17:49	☆				
setup.sh	805 octets	Today 17:44	☆				
tsconfig.json	702 octets	Today 17:34	☆				
.env	76 octets	Today 17:14	☆				

8. Prisma → @mostajs/orm migration (bootstrap)

```
hmd@amia26: ~/DEMO-00/ecom-saas-prisma
✓ Ready in 801ms
⚠ Warning: Next.js inferred your workspace root, but it may not be correct.
We detected multiple lockfiles and selected the directory of /home/hmd/package-lock.json as the root directory.
To silence this warning, set 'turboPack.root' in your Next.js config, or consider removing one of the lockfiles if it's not needed.
See https://nextjs.org/docs/app/api-reference/config/next-config-js/turboPack#root-directory for more information.
Detected additional lockfiles:
  * /home/hmd/DEMO-00/ecom-saas-prisma/package-lock.json

○ Compiling / ...
GET / 200 in 6.3s (next.js: 5.8s, application-code: 453ms)
GET /api/tenants 200 in 4.4s (next.js: 3.3s, application-code: 1143ms)
✓ Finished writing to filesystem cache in 56s
✓ Finished writing to filesystem cache in 27.8s
GET / 200 in 129ms (next.js: 16ms, application-code: 113ms)
GET /api/products 200 in 1239ms (next.js: 209ms, application-code: 1030ms)
GET /api/orders 200 in 1218ms (next.js: 207ms, application-code: 1011ms)
GET /api/users 200 in 1267ms (next.js: 300ms, application-code: 967ms)
cccc^C
hmd@amia26:~/DEMO-00/ecom-saas-prisma$ npx @mostajs/orm-cli bootstrap
```

Install

```
npm install @mostajs/orm
# + the driver for your dialect :
npm install better-sqlite3      # or: pg, mysql2, mongoose, oracledb, mssql, ibm_db, mariadb, @sap/hana-client, @google-cloud/spanner
npm install sql.js              # SQLite in the browser / Bolt.new / Workers - no native binary (dialect: 'sqljs')
npm install @electric-sql/pglite # PostgreSQL in the browser - idb:// persistence (dialect: 'pglite')
```

Define a schema

```
import type { EntitySchema } from '@mostajs/orm'

export const UserSchema: EntitySchema = {
  name: 'User',
  collection: 'users',
  timestamps: true,
  fields: {
    email: { type: 'string', required: true, unique: true },
    name: { type: 'string', required: true },
  },
  relations: {
    roles: { target: 'Role', type: 'many-to-many', through: 'user_roles' },
  },
  indexes: [{ fields: { email: 'asc' }, unique: true }],
}
```

Unique keys

A field can be marked unique, or several fields can be combined into a composite unique constraint via [indexes](#).

```
export const MemberSchema: EntitySchema = {
  name: 'Member',
  collection: 'members',
  fields: {
    email: { type: 'string', required: true, unique: true }, // single unique
    tenantId: { type: 'string', required: true },
    slug: { type: 'string', required: true },
  },
  indexes: [
    { fields: { tenantId: 'asc', slug: 'asc' }, unique: true }, // composite unique
  ],
}
```

Both shapes enforce a DDL `UNIQUE` constraint on SQL dialects and a unique index on MongoDB. Lookup works the same way :

```
await repo.findOne({ email: 'a@b.com' }) // single unique
await repo.findOne({ tenantId: 't1', slug: 'admin' }) // composite unique
```

Connect & CRUD

```
import { registerSchemas, getDialect, BaseRepository } from '@mostajs/orm'

registerSchemas([UserSchema])
const dialect = await getDialect() // reads DB_DIALECT + SGBD_URI from env
const repo = new BaseRepository(UserSchema, dialect)

await repo.create({ email: 'a@b.com', name: 'Admin' })
await repo.findOne({ email: 'a@b.com' })
await repo.findAll({ status: 'active' }, { sort: { name: 1 }, limit: 10 })
await repo.update(id, { name: 'Updated' })
await repo.delete(id)
await repo.findByIdWithRelations(id, ['roles'])
await repo.upsert({ email: 'a@b.com' }, { name: 'Upserted' })
await repo.count({ status: 'active' })
```

Query options

```
await repo.findAll(
  { status: 'active', role: { $in: ['admin', 'editor'] } }, // filter
  {
    sort: { createdAt: -1, name: 1 }, // multi-field sort
    skip: 20, limit: 10, // pagination
    select: ['id', 'email', 'name'], // projection
  },
)
```

`$in`, `$nin`, `$gt`, `$gte`, `$lt`, `$lte`, `$ne`, `$regex`, `$exists` are all supported uniformly across SQL & NoSQL dialects.

Soft delete (native)

Set `softDelete: true` on a schema and `@mostajs/orm` handles everything :

```
export const PostSchema: EntitySchema = {
  name: 'Post',
  collection: 'posts',
  timestamps: true,
  softDelete: true, // ← that's it
  fields: { title: { type: 'string' }, body: { type: 'string' } },
}
```

```
await postRepo.delete(id) // sets deletedAt + isDeleted=true
await postRepo.findAll({}) // excludes soft-deleted rows
await postRepo.findAll({}, { includeDeleted: true }) // include them
await postRepo.restore(id) // un-delete (clears deletedAt)
await postRepo.purge(id) // hard-delete, row removed for good
```

Works identically across SQL (column `deletedAt timestamp NULL`) and MongoDB (field `deletedAt` indexed). No more home-rolled `deleted` flag mismatches — the **R003-SOFT-DELETE-INCONSISTENT** validator rule will flag manual patterns (and auto-migrate them via `--fix R003`).

Transactions

Group multiple operations into a single atomic unit. SQL dialects (PostgreSQL, MySQL/MariaDB, SQLite, SQL Server, Oracle, DB2, CockroachDB, HANA, Sybase, HSQLDB, Spanner) wrap the callback in `BEGIN` / `COMMIT` / `ROLLBACK`. If any operation throws, every write inside the block is rolled back.

```
import { getDialect } from '@mostajs/orm'

const dialect = await getDialect()

await dialect.$transaction(async (tx) => {
  await tx.create('accounts', { id: 'a', balance: 100 })
  await tx.update('accounts', { id: 'b' }, { $inc: { balance: -50 } })
  await tx.update('accounts', { id: 'a' }, { $inc: { balance: 50 } })
  // throw here → both updates are rolled back, `accounts.a` row is removed
})
```

Isolation : default per dialect (SQL → `READ COMMITTED`, SQLite → `DEFERRED`). Pass `{ isolation: 'SERIALIZABLE' }` as 2nd argument to override (SQL only).

All SQL dialects listed above support ACID natively — PostgreSQL, MySQL/MariaDB, SQL Server, Oracle, DB2, SQLite, CockroachDB, HANA, Sybase, HSQLDB, Spanner. No configuration required beyond the usual connection.

MongoDB is the only exception : multi-document ACID transactions require a replica set (a single-node `mongod --replSet rs0` is enough for dev — this is a MongoDB server requirement, not a limitation of this library). On a standalone server, `$transaction` runs the callback without wrapping — safe for read-heavy flows, non-atomic for writes.

Manual transactions — `beginTransaction` / `commitTx` / `rollbackTx` (v1.11+)

When the `$transaction(cb)` callback pattern is too restrictive (transaction spans several unrelated functions, commit depends on an external event), use the manual API :

```
const tx = await dialect.beginTransaction()
try {
  await dialect.create(UserSchema, { email: 'a@b.c', name: 'A' })
  await someExternalCheck() // could be async, could take seconds
  if (ok) await dialect.commitTx(tx)
  else await dialect.rollbackTx(tx)
} catch (e) {
  await dialect.rollbackTx(tx)
  throw e
}
```

Nested transactions — SAVEPOINTS are used automatically :

```

const outer = await dialect.beginTransaction() // → BEGIN
await dialect.create(UserSchema, { email: 'o@x.io', name: 'Outer' })

const inner = await dialect.beginTransaction() // → SAVEPOINT mosta_sp_2_xxxx
await dialect.create(UserSchema, { email: 'i@x.io', name: 'Inner' })
await dialect.rollbackTx(inner) // → ROLLBACK TO SAVEPOINT
//                               (inner row gone, outer untouched)

await dialect.commitTx(outer) // → COMMIT
//                               (outer row persisted)

```

Depth unbounded as long as the engine supports `SAVEPOINT` (every SQL dialect above except **Spanner**). MSSQL / Sybase use `SAVE TRANSACTION` / `ROLLBACK TRANSACTION` internally — transparent to the API. `commitTx` / `rollbackTx` enforce LIFO order (out-of-order commit throws, out-of-order rollback is silent).

Environment

```

DB_DIALECT=postgres
SGBD_URI=postgresql://user:pass@localhost:5432/mydb
DB_SCHEMA_STRATEGY=update # validate | update | create | create-drop | none
DB_SHOW_SQL=true

```

The naming mirrors Hibernate's `hibernate.hbm2ddl.auto` / `hibernate.show_sql` properties (see [Hibernate User Guide § schema strategies](#)). Values have identical semantics: `validate` / `update` / `create` / `create-drop` / `none`.

The dialect matching `DB_DIALECT` is **lazy-loaded at runtime** (v1.9.3+). Only the driver you actually use is evaluated — no other dialect module enters your bundle. This is what makes `@mostajs/orm` safe to pull into a Next.js / Vite / SvelteKit project without bundler workarounds.

Profile cascade with `MOSTA_ENV` (v1.13+)

Powered by `@mostajs/config`. Keep **one** `.env` file with profile-prefixed overrides and switch via a single `MOSTA_ENV` variable — exactly like [Spring Boot profiles](#) (`spring.profiles.active=test` loading `application-test.properties`).

```

# .env - committed (non-secret) defaults
MOSTA_ENV=TEST

# Base defaults (used when no profile, or as fallback)
DB_DIALECT=sqlite
SGBD_URI=./data.sqlite

# Profile overrides
TEST_DB_DIALECT=sqlite
TEST_SGBD_URI=./test.sqlite
TEST_DB_SCHEMA_STRATEGY=create-drop

DEV_DB_DIALECT=postgres
DEV_SGBD_URI=postgres://localhost:5432/devdb
DEV_DB_SCHEMA_STRATEGY=update

PROD_DB_DIALECT=mongodb
PROD_SGBD_URI=${SCALEWAY_MONGO_URI} # secret injected by orchestrator
PROD_DB_SCHEMA_STRATEGY=validate

```

Resolution cascade (first non-empty wins) :

1. `${MOSTA_ENV}_${KEY}` — profile-prefixed
2. `${KEY}` — plain
3. `fallback` argument
4. `undefined` — no crash, caller decides whether that's fatal

Silent fallback is guaranteed : a missing profile override never throws, it just falls through to the plain variable or to the default. Empty strings (`TEST_DB_DIALECT=`) are treated as “not set” so they don't silently leak a blank value.

For generic use outside `@mostajs/orm`, import directly from the config package :

```

import { getEnv, getEnvBool, getEnvNumber, getCurrentProfile } from '@mostajs/config'

const url = getEnv('REDIS_URL', 'redis://localhost:6379')
console.log(`Profile : ${getCurrentProfile() ?? 'none'}`)

```

Switch databases with one env var

```
DB_DIALECT=sqlite      SGBD_URI=./data.sqlite
DB_DIALECT=postgres    SGBD_URI=postgres://...
DB_DIALECT=mongodb     SGBD_URI=mongodb://...
# same code in both cases
```

Multi-app DB cohabitation — DB_TABLE_PREFIX (v2.3.0+)

Hibernate-style `physical_naming_strategy` for `@mostajs/orm`. Lets several apps share one physical database without colliding on common names like `users`, `roles`, `permissions`, etc. — particularly useful on **Oracle / MSSQL / HANA** where the SQL schema is bound to the connection user (so two apps using the same DB user otherwise silently share tables).

```
# .env
DB_TABLE_PREFIX=mp_
```

```
// or via API
await createConnection(
  { dialect: 'oracle', uri: '...', tablePrefix: 'mp_' },
  schemas,
)
```

What gets prefixed at runtime (the developer keeps writing `collection: 'users'` everywhere — the prefix lives in the dialect layer) :

- `CREATE / DROP / ALTER TABLE`
- `CREATE INDEX`
- `FOREIGN KEY REFERENCES`
- Junction tables (`RelationDef.through`)
- `FROM / INSERT / UPDATE / DELETE`
- Mongo collection physical name (via `mongoose.model(name, schema, prefixed)`)

If `tablePrefix` is `undefined` or empty, behavior is strictly identical to 2.2.x — fully backward-compatible.

i **Relation name ≠ table name** — `findByIdWithRelations(id, ['roles'])` uses the **logical** relation name declared in `EntitySchema.relations.roles`. The physical join table comes from `RelationDef.through` (e.g. `'user_roles'`) and that name is the one prefixed (→ `mp_user_roles`). No physical table name is hard-coded anywhere in `@mostajs/*` libraries — everything routes through the schemas + the dialect's `getPrefixedName()` .

A runnable showcase of `DB_TABLE_PREFIX` lives in `@mostajs/orm-samples` [sample 16](#) (`mosta-parkmanager`).

Subpaths

Subpath	When to use
<code>@mostajs/orm</code>	The core ORM API : <code>getDialect</code> , <code>registerSchemas</code> , <code>BaseRepository</code> , <code>EntityService</code> , schema types, <code>diffSchemas</code> , errors.
<code>@mostajs/orm/bridge</code>	JDBC bridge (v1.9.4+) : <code>JdbcNormalizer</code> , <code>BridgeManager</code> , <code>JDBC_REGISTRY</code> , jar upload. Pulled out of the root to keep <code>child_process</code> / <code>fs</code> spawn out of client bundles.
<code>@mostajs/orm/register</code>	Zero-code registration side-effect for dynamic schema loading.
<code>@mostajs/orm/validator</code>	v1.14+ — ORMConceptValidator : algorithmic linter for <code>EntitySchema</code> sets. Detects 18 conceptual anomalies (empty relations, FK naming inconsistency, soft-delete patterns, dead code, missing audit, unbounded blobs...). See below.

ORMConceptValidator (v1.14+)

Algorithmic linter for your ORM schemas — detects 18 conceptual anomalies before they bite in production. Zero IA, zero heuristics flou, **fully generic** (no hardcoded entity name — `KNOWN_ENTITY_REFS` is derived at runtime from the schemas you pass).

Real-world impact (v1.17.0) : applied to iquesta (21 schemas, 70 findings) — `--fix R001,R001B,R002,R003` auto-corrected all 16 structural anomalies in seconds. Cross-projects calibration over 17 mostajs/* + apolocene codebases : **247 findings** identified, **-23** after applying C1+C2 to iquesta alone.

Quick start

```
# CLI – point it at your schemas directory
npx mostajs-orm-validator ./schemas

# With cross-file rules (R005, R007, R008, R011, R012, R014, R015) :
npx mostajs-orm-validator ./schemas --src ./lib

# In a CI pipeline :
npx mostajs-orm-validator ./schemas --src ./lib --ci --max-warnings 0
```

Or programmatically :

```
import { validateSchemas, formatText } from '@mostajs/orm-validator'
import * as schemas from './schemas'

const report = await validateSchemas(Object.values(schemas), {
  sourceRoot: './lib',
})

console.log(formatText(report))
console.log(`${report.findings.length} findings`)
```

What it detects (18 rules)

ID	Severity	Detection
R001-EMPTY-RELATIONS	warning	String field named like another entity (e.g. <code>project</code> , <code>respondent</code>) but <code>relations: {}</code> empty → loses ORM cascade & FK validation
R002-FK-NAMING-INCONSISTENT	warning	Mix of conventions in same set (<code>parentId</code> vs <code>project</code> , <code>questionId</code> vs <code>section</code>) — flags the minority
R003-SOFT-DELETE-INCONSISTENT	warning/info	Multiple soft-delete patterns concurrent (<code>deleted</code> / <code>cancelled</code> / <code>archived</code>) OR manual <code>deleted/deletedAt</code> while <code>softDelete: true</code> is available natively
R004-DUPLICATE-ENTITY-SHAPE	info	Pair of schemas with Jaccard on field names ≥ threshold (default 0.7) — possible legacy
R004B-LEGACY-ENTITY	info/warning	Name overlap (substring ≥ 4 chars or Jaro-Winkler ≥ 0.75) — flags the smaller schema. Bumps to warning if <code>legacy/deprecated</code> comment found in sources
R005-ANY-TYPED-REPO	warning	<code>BaseRepository<any></code> in source files — typing lost. Needs <code>--src</code>
R006-JSON-AS-RELATION	info	<code>*sJson</code> field containing list of FK slugs/ids — should be normalized into junction table
R007-REDUNDANT-DERIVED-FIELD	info	Persisted field duplicate of a pure function of its id (e.g. <code>blobPath</code> derivable from <code>archiveBlobPath(id)</code>). Needs <code>--src</code>
R008-BEST-EFFORT-FK-RESOLVER	warning	<code>best-effort</code> / <code>TOD0 V2</code> / <code>HACK</code> comment + <code>??</code> <code>null</code> fallback → root cause hidden. Needs <code>--src</code>
R009-MISSING-LOOKUP-INDEX	info/hint	<code>unique</code> field without dedicated index, OR FK string without index for inverse lookups
R010-MISSING-AUDIT-TABLE	hint	No schema resembling <code>AuditLog</code> (actor + action + timestamp) — sensitive actions untraceable
R011-LEGACY-DEAD-CODE	info	TS source file never imported (entry points like <code>page.tsx</code> / <code>route.ts</code> excluded). Needs <code>--src</code>
R012-DUPLICATE-IMPLEMENTATION	info	Pair of source files exporting overlapping function signatures (Jaccard ≥ 0.85). Needs <code>--src</code>
R013-MISSING-CASCADE	warning	<code>many-to-one</code> relation without explicit <code>onDelete</code> → orphans on parent delete
R014-REPO-FACTORY-BOILERPLATE	info	≥ 5 <code>get*Repo()</code> helpers in same file — suggest factory. Needs <code>--src</code>
R015-FLAT-LIB-STRUCTURE	hint	Directory with > 25 flat files — suggest sub-directory organisation. Needs <code>--src</code>
R016-AUDIT-EMAIL-AS-STRING	info	<code>createdBy</code> / <code>validatedBy</code> /etc. typed string instead of FK User → loses ref. integrity if email changes
R017-UNBOUNDED-BLOB-FIELD	hint	<code>*Json</code> / <code>*Payload</code> / <code>*Blob</code> / <code>*Manifest</code> without documented size limit
R018-EXTERNAL-SCHEMA-OVERSCOPED	info	(<i>stub V2 — full impl in V3 with ts-morph</i>) External schema with many unused fields

Output formats

```
# Console output (TTY-aware ANSI colors)
npx mostajs-orm-validator ./schemas

# JSON (for CI / diff)
npx mostajs-orm-validator ./schemas --format json --out report.json

# Markdown (human-readable report)
npx mostajs-orm-validator ./schemas --format markdown --out REPORT.md
```

Example output :

```
x Section.project          R001-EMPTY-RELATIONS          warning
  Field 'Section.project' looks like an FK to 'Project' but no ORM
  relation declared.
  Suggestion:
    relations: {
      project: { type: 'many-to-one', target: 'Project',
                required: true, onDelete: 'cascade' },
    },
```

Configuration

All thresholds and patterns are configurable — no hardcoded business strings. Pass a config to `validateSchemas` :

```
const report = await validateSchemas(schemas, {
  sourceRoot: './lib',
  ignore: ['R015', 'R017'], // skip these rules entirely
  rules: { R001: 'error' }, // override severity (e.g. block CI on R001)
  softDeletePatterns: [
    { flag: 'deleted', timestamp: 'deletedAt' },
    { flag: 'cancelled', timestamp: 'cancelledAt' },
    { flag: 'archived', timestamp: 'archivedAt' },
    // add your project-specific patterns here
  ],
  auditByFields: ['createdBy', 'validatedBy', 'reviewedBy'],
  thresholds: {
    duplicateEntityJaccard: 0.7, // R004
    duplicateImplJaroWinkler: 0.85, // R012
    flatLibMaxFiles: 25, // R015
  },
})
```

CI integration

```
// package.json
{
  "scripts": {
    "lint:schemas": "mostajs-orm-validator ./schemas --src ./lib --ci --max-warnings 0"
  }
}
```

The `--ci` flag exits with code 1 if the number of `error + warning` findings exceeds `--max-warnings` (default 0). Bind it to your pre-commit hook or GitHub Actions to block regressions.

Auto-fix (v1.15+ — V3-A) — `--fix` workflow

The validator can **apply the fix it suggests**, in-place via [ts-morph](#), for a subset of rules :

Rule	Auto-fix action
R001-EMPTY-RELATIONS	Move the FK string field out of <code>fields: {}</code> and add a matching <code>many-to-one</code> entry to <code>relations: {}</code> with <code>onDelete: 'cascade'</code> .
R001B-FIELD-RELATION-DUPLICATE	Remove the redundant field when both <code>field</code> (<i>string</i>) and <code>relations.field</code> (<i>many-to-one</i>) exist for the same FK — leftover of a partial earlier fix. (v1.17+)
R002-FK-NAMING-INCONSISTENT	Rename the field in the schema to match the majority convention (<code>parentId</code> → <code>parent</code>). Cross-file consumer rename is left to the dev (use your IDE rename refactor).
R003-SOFT-DELETE-INCONSISTENT	Add <code>softDelete: true</code> and remove manual <code>deleted</code> + <code>deletedAt</code> fields. (v1.17+)
R016-AUDIT-EMAIL-AS-STRING	Convert the string field (<code>createdBy</code> , <code>updatedBy</code> ...) into a <code>many-to-one</code> relation to <code>User</code> with <code>onDelete: 'set-null'</code> .


```

# Dry-run — show diffs without writing
npx mostajs-orm-validator ./schemas --fix-dry-run

# Apply — writes <file>.bak backups by default
npx mostajs-orm-validator ./schemas --fix

# Apply only a subset of rules
npx mostajs-orm-validator ./schemas --fix --fix-rules R001,R003

# Without .bak files (CI / git-tracked workflow)
npx mostajs-orm-validator ./schemas --fix --no-backup

# Roll back the last --fix run (restores every <file>.bak)
npx mostajs-orm-validator ./schemas --rollback-fix

```

Workflow recommended :

1. Commit your current state (so you have a clean diff baseline).
2. `--fix-dry-run` first — review the proposed diffs.
3. `--fix` once you're confident.
4. Run your test suite. If something broke, `--rollback-fix` restores the `.bak` files. Iterate with `--fix-rules` to apply only what works.
5. `git diff` + commit.

In-process API

```

import {
  validateSchemas,
  applyFixes,
  rollbackFixes,
  formatText, formatJson, formatMarkdown,
} from '@mostajs/orm/validator'

const report = await validateSchemas(schemas, { sourceRoot: './lib' })
console.log(formatJson(report, true)) // pretty JSON for CI artifact

const fixResults = await applyFixes(report, {
  sourceRoot: './schemas',
  dryRun: false,
  rules: ['R001', 'R001B', 'R003'], // narrow the scope
  backup: true,
})

console.log(`Applied ${fixResults.filter(r => r.applied).length} fixes`)

// Tests fail ? roll back :
rollbackFixes('./schemas') // restores all .bak files and deletes them

```

Resilience features (v1.17+)

- **Cascade ts-morph mitigation** : when two fixes target the same file (e.g. `registration.schema.ts` with both `RegistrationSchema` and `AttendanceSchema`), the fixer reloads the `SourceFile` between fixes via `removeSourceFile + createSourceFile` to avoid "node forgotten" crashes.
- **Text fallback** : if ts-morph `.remove()` crashes on an end-of-line comment, the fixer falls back to a robust regex that removes the field cleanly (e.g. `project: { type: 'string' }, // FK Project`).
- **Try / catch around each fix** : a crash on one finding never aborts the rest of the batch. Failures are reported in `skipped` with a `reason`.

VSCode extension (v0.2.0)

A VSCode extension wraps the validator so you get **inline squiggles** on your `*.schema.ts` files plus **Code Actions** (*quick-fix UI*) for the auto-fixable rules :

```

# Install from VSIX (until Marketplace publish)
code --install-extension mostajs-orm-vscode-0.2.0.vsix

```

Features :

- In-process import of `@mostajs/orm/validator` (no CLI spawn → faster).
- Debounced 300ms re-lint on save / edit.
- Hover : full finding details + suggestion preview.
- Code Action ("💡 lightbulb") : applies the corresponding `--fix` rule in-place. The `.bak` discipline applies — undo via `--rollback-fix` or VSCode "Undo Local Changes".

Source : `mostajs/mosta-orm-vscode`.

Generic by design

The validator is **fully generic** — no hardcoded entity name, no project-specific assumption. The set of “known entities” (`KNOWN_ENTITY_REFS`) is derived at runtime from the schemas you pass. Same binary detects the same anti-patterns in any consumer codebase.

TypeScript schemas

The CLI loads `.ts / .tsx / .js / .mjs` files directly via `jiti` — no pre-compile step required. TypeScript `paths` aliases are resolved automatically.

Recipes (cookbook)

Pagination + total count

```
const [rows, total] = await Promise.all([
  postRepo.findAll({ author: userId }, { sort: { createdAt: -1 }, skip: 20, limit: 10 }),
  postRepo.count({ author: userId }),
])
```

Composite unique upsert (*idempotent seed*)

```
// Reuses the {tenantId, slug} composite unique to avoid race conditions
await memberRepo.upsert(
  { tenantId: 't1', slug: 'admin' },
  { tenantId: 't1', slug: 'admin', email: 'admin@t1.io', role: 'owner' },
)
```

Many-to-many through junction (*read + insert*)

```
// Schema : User.roles = many-to-many → Role through 'user_roles'
const user = await userRepo.findByIdWithRelations(userId, ['roles'])
console.log(user.roles) // [{ name: 'admin' }, ...]

// Add a role link (writes into the junction table) :
await dialect.linkRelation('User', userId, 'roles', roleId)
await dialect.unlinkRelation('User', userId, 'roles', roleId)
```

Cross-dialect bootstrap (*one codebase, two environments*)

```
// boot.ts – runs identically against SQLite (dev) and PostgreSQL (prod)
import { registerSchemas, getDialect } from '@mostajs/orm'
import { schemas } from './schemas'

export async function boot() {
  registerSchemas(schemas)
  const dialect = await getDialect() // picks env at runtime
  await dialect.initSchema(schemas) // DDL per strategy
  return dialect
}

// dev : DB_DIALECT=sqlite SGBD_URI=./data.sqlite npm run boot
// prod: DB_DIALECT=postgres SGBD_URI=$DATABASE_URL npm run boot
```

Transaction with isolation upgrade

```
await dialect.$transaction(
  async (tx) => {
    const acct = await tx.findOne('accounts', { id: 'a' })
    if (acct.balance < 100) throw new Error('insufficient')
    await tx.update('accounts', { id: 'a' }, { $inc: { balance: -100 } })
    await tx.create('ledger', { type: 'debit', amount: 100, accountId: 'a' })
  },
  { isolation: 'SERIALIZABLE' }, // upgrade isolation
)
```

Lint your schemas in a pre-commit hook

```
# .husky/pre-commit
#!/bin/sh
npx mostajs-orm-validator ./schemas --src ./lib --ci --max-warnings 0
```

Soft-delete migration (*legacy to native*)

```
# Detects manual deleted/deletedAt pattern, suggests softDelete: true natif
npx mostajs-orm-validator ./schemas --fix-dry-run --fix-rules R003

# Apply when satisfied :
npx mostajs-orm-validator ./schemas --fix --fix-rules R003
```

The validator will :

1. Add `softDelete: true` to the schema object.
2. Remove the manual `deleted` and `deletedAt` fields from `fields: {}`.
3. Leave a `.bak` backup so you can `--rollback-fix` if your runtime code relied on the manual fields directly.

EntityService (for @mostajs/net)

```
import { EntityService } from '@mostajs/orm'

const service = new EntityService(dialect)
const res = await service.execute({
  op: 'findAll',
  entity: 'User',
  filter: { status: 'active' },
  relations: ['roles'],
  options: { limit: 10 },
})
```

Operations : `findAll`, `findOne`, `findById`, `create`, `update`, `delete`, `deleteMany`, `count`, `search`, `aggregate`, `upsert`, `updateMany`, `addToSet`, `pull`, `increment`.

Schema management

```
await dialect.initSchema(getAllSchemas()) // create / update DDL per strategy
await dialect.truncateTable?.('users')
await dialect.truncateAll?.(getAllSchemas())
await dialect.dropTable?.('users')
await dialect.dropSchema?.(getAllSchemas())
await dialect.dropAllTables?.()
```

Dialect-level guarantees (v1.13+)

Two classes of correctness fixes ship with 1.13, both driven by real production pain encountered during `@mostajs/replicator` runs.

SQL dialects (`AbstractSqlDialect`)

- **FK columns preserve falsy-but-valid values** (`0`, `false`). The previous short-circuit `data[name] || null` silently replaced legitimate zero IDs and boolean `false` with `null`, breaking FK writes whose source-side PK happened to be `0`. The insert/update path now uses `value === '' ? null : (value ?? null)` — empty strings still null-out (SQL foreign-key constraints reject them on most dialects) but numeric zero, `false` and any non-empty value round-trip intact.
- **One-to-one relations get a column-level `UNIQUE` constraint**. Emitted both at `CREATE TABLE` time and at `ALTER TABLE ADD` time when growing an existing schema. Matches the JPA / Hibernate semantics where an `@OneToOne` FK must be injective (otherwise the “one” side of the relation is not actually single-valued).

Mongo dialect

- **FK fields accept UUID strings in addition to native `ObjectId`**. `buildMongooseSchema` now declares FK refs as `Schema.Types.Mixed` rather than `Schema.Types.ObjectId`. Replicated documents originating from a SQL dialect (SQLite / Postgres / ... using UUID primary keys) are no longer rejected by Mongoose path validation. A native Mongo app writing proper `ObjectId` refs keeps working unchanged.
- **`findAll()` / `findOne()` fall back to `{ id: fkValue }` when `populate()` returns `null`**. When a UUID-string FK cannot be resolved through the default Mongoose `_id` lookup (which expects matching type), the dialect keeps a raw `lean()` query alongside the populated one and patches the missing refs post-hoc by a direct `findOne({ id: fk })` on the target collection. Transparent to the caller, prevented silent data loss during cross-dialect reads.

Together, these four items unblock bidirectional SQL ↔ Mongo sync through `@mostajs/replicator`.

Multiple simultaneous connections

```
import { createIsolatedDialect, registerNamedConnection, getNamedConnection } from '@mostajs/orm'

const oracle = await createIsolatedDialect({ dialect: 'oracle', uri: '...' }, [UserSchema])
const mongo = await createIsolatedDialect({ dialect: 'mongodb', uri: '...' }, [AuditLogSchema])
registerNamedConnection('audit', mongo)

// Later, anywhere in the codebase :
const conn = getNamedConnection('audit')
```

Ecosystem

Package	Description
@mostajs/orm-bridge	Keep your Prisma code, run it on any of the 13 databases (<code>createPrismaLikeDb()</code> is a drop-in replacement for <code>new PrismaClient()</code>).
@mostajs/orm-cli	<code>npx @mostajs/orm-cli</code> — interactive CLI : convert schemas, init databases, scaffold services, replicator + monitor, seeding, bootstrap Prisma migration.
@mostajs/orm-adapter	Convert Prisma / JSON Schema / OpenAPI / native <code>.mjs</code> to <code>EntitySchema[]</code> (bidirectional).
@mostajs/replicator	Cross-dialect replication : CQRS master/slave, CDC rules (snapshot + incremental), wildcard <code>*</code> , failover (<code>promoteToMaster</code>). As of <code>@mostajs/orm v1.13</code> , Mongo FK columns accept UUID strings coming from SQL dialects (populate falls back to <code>{ id: uuid }</code> lookup).
@mostajs/orm-copy-data	Cross-dialect data copy : 1 source (DB / CSV / JSON / SQL dump) → N destinations. Backup, migration, seeding. CLI (<code>mostajs-copy</code>) + API. Cron-ready.
@mostajs/replica-monitor	Live web dashboard — replicas status, CDC stats, activity stream. Zero DB connections (reads tree + stats files).
@mostajs/media	Screen capture + video editor (split, speed, stickers, subtitles) + server-side ffmpeg export + project persistence (ORM + SQLite).
@mostajs/config	Env loader with <code>MOSTA_ENV</code> profile cascade (Spring-Boot-style). Used by <code>orm/auth/payment/music</code> .

Design inspirations

`@mostajs/orm` draws from three decades of mature ORM engineering in the Java ecosystem, adapted to the TypeScript / Node.js runtime :

Borrowed concept	Source	@mostajs/orm equivalent
<code>SessionFactory</code> / <code>EntityManagerFactory</code>	Hibernate · JPA	<code>getDialect()</code> returning a cached singleton
Entity metadata (annotations / XML)	Hibernate , JPA <code>@Entity</code>	<code>EntitySchema</code> — declarative TypeScript schema
<code>@OneToMany</code> / <code>@ManyToMany</code> / <code>@OneToOne</code> / <code>@ManyToMany</code>	JPA	<code>relations: { ..., type: 'one-to-many' \ ... }</code>
<code>@JoinColumn</code> , <code>@JoinTable</code>	JPA	<code>joinColumn</code> , <code>through</code>
<code>CascadeType</code> / <code>FetchType</code>	JPA	<code>cascade</code> , <code>fetch</code> in <code>RelationDef</code>
Cascade types (<code>PERSIST</code> , <code>REMOVE</code> , <code>ALL</code>)	JPA	<code>cascade: ['persist', 'remove', 'all']</code>
Schema-generation strategies (<code>validate</code> , <code>update</code> , <code>create</code> , <code>create-drop</code>)	Hibernate <code>hibernate.hbm2ddl.auto</code>	<code>DB_SCHEMA_STRATEGY</code> (same names, same semantics)
Show-SQL / format-SQL / highlight-SQL	Hibernate (<code>hibernate.show_sql</code> , <code>hibernate.format_sql</code>)	<code>DB_SHOW_SQL</code> , <code>DB_FORMAT_SQL</code> , <code>DB_HIGHLIGHT_SQL</code>
<code>SAVEPOINT</code> for nested transactions	SQL standard, JPA spec	<code>beginTx()</code> inside <code>beginTx()</code> emits <code>SAVEPOINT</code>
Repository pattern	Spring Data	<code>BaseRepository<T></code> with typed CRUD
Profile-based configuration	Spring Boot profiles (<code>spring.profiles.active=test</code>)	<code>MOSTA_ENV=TEST</code> + <code>TEST_KEY=value</code> cascade (via <code>@mostajs/config</code>)
Environment-aware externalized config	Spring Boot <code>application-\${profile}.properties</code>	One <code>.env</code> with <code>\${PROFILE}_\${KEY}</code> overrides

Why borrow from the Java ecosystem ?

Hibernate (2001), JPA (2006, JSR 220), Spring Data (2008), Spring Boot (2014) have collectively survived two decades of production workloads. Their vocabulary and semantics are **industry defaults** : developers who have worked with any of them recognize `@OneToMany`, `CascadeType.ALL`, `spring.profiles.active`, `hibernate.hbm2ddl.auto=update`, `SAVEPOINT`, etc. immediately. Reusing those names in `@mostajs/orm` cuts the learning curve and avoids inventing a parallel dialect.

Further reading :

- [Hibernate ORM](https://hibernate.org/orm/documentation/) — <https://hibernate.org/orm/documentation/>
- [Jakarta Persistence \(JPA\)](https://jakarta.ee/specifications/persistence/) — <https://jakarta.ee/specifications/persistence/>
- [Spring Framework](https://spring.io/projects/spring-framework) — <https://spring.io/projects/spring-framework>
- [Spring Data](https://spring.io/projects/spring-data) — <https://spring.io/projects/spring-data>
- [Spring Boot profiles](https://docs.spring.io/spring-boot/reference/features/profiles.html) — <https://docs.spring.io/spring-boot/reference/features/profiles.html>
- [Spring Boot externalized configuration](https://docs.spring.io/spring-boot/reference/features/external-config.html) — <https://docs.spring.io/spring-boot/reference/features/external-config.html>

License

AGPL-3.0-or-later + commercial license available.

For closed-source commercial use : drmdh@msn.com

Author

Dr Hamid MADANI drmdh@msn.com