The Plain Web Stack

TradJS, sqlite-zod-orm and measure-fn: what they are, how they are tested, and when to use them

webstack, November 2026 · 2026-11-18

Contents

The case in one page

Why plain

TradJS

- Files are routes
- Middleware hands typed values to pages
- Browser scripts mount where needed
- Security that is on by default
- Static pages

sqlite-zod-orm

- One schema language
- Typed reads, exact writes
- Hooks that respect transactions
- Live changes across processes

measure-fn

How we test

What the tests found

- TradJS
- sqlite-zod-orm
- measure-fn

Speed

Limits and what is next

Start in ten minutes

Compared with what you know

- The page: choose a rendering framework
- The database: choose a schema and persistence model
- The request: choose logs or traces
- Make the choice

The case in one page

Three small open-source libraries for Bun make one stack:

Layer	Library	What it replaces
Web framework	**TradJS** (`tradjs`)	Next.js, Remix, a client router, a bundler config
Database	**sqlite-zod-orm** (`sqlite-zod-orm`)	Prisma or Drizzle plus a database server
Logging and timing	**measure-fn** (`measure-fn`)	a logger plus a tracing SDK

The idea is to **do less, and prove what you do**.

- Pages are rendered on the server and the browser navigates with real links.
- Data lives in one SQLite file, described by the same Zod schemas that validate it.
- Every meaningful step prints one line with its time.
- Then every claim is checked by running the real thing: a real browser, a real database file built by an older version, real processes.

Use it when:

- you are building a product with pages, forms, accounts and an API;
- one machine (or a few processes on one machine) serves your traffic;
- you want to read every layer of your stack in an afternoon.

Do not use it when:

- you need a database server shared by many machines (SQLite is one file on one disk);
- your app is a single-page application that never reloads;
- your team depends on the React ecosystem's component libraries. TradJS uses its own small JSX renderer, not React.

> Note: every number in this book comes from a test that was run on 18 November 2026 and is recorded in the libraries' changelogs. Where something was not measured, the book says so.

Next: read chapter 2 for the reasoning, or jump to chapter 10 to start an app.

Why plain

A browser already does most of what a modern front-end framework rebuilds:

- it follows links and submits forms;
- it keeps history, restores scroll position and caches pages for the back button;
- it shows a page as soon as its HTML arrives.

Client-side routers rebuild all of that in JavaScript, and then spend more code fixing the parts they got wrong (scroll, focus, history, loading states).

TradJS takes the other path:

1. **Every page is HTML from the server.** A link is a real link and a form is a real form post. A page that needs no interaction ships no JavaScript.
2. **Interaction is local.** A page that needs a counter, a live list or a drag-and-drop area gets a `page.client.tsx` file next to it. That script mounts into one element.
3. **Navigation is smooth anyway.** Cross-document view transitions, a CSS feature in current browsers, animate ordinary page loads, so leaving and arriving feel instant without a client router.
4. **The back button is the browser's.** Pages come back from the back/forward cache. A page script that returns a cleanup function is stopped when the page is hidden and started again when it returns.

The same reasoning applies to data. Most apps that run on one machine do not need a database server. SQLite is a file, it is fast, and it runs inside your process. What it lacks is a typed, validated interface, and **sqlite-zod-orm** adds exactly that. The Zod schema that validates a form also defines the table.

And to logging: instead of a logging library plus a tracing SDK plus a dashboard, **measure-fn** prints one line per measured step, nested under its parent, with its duration.

Next: chapters 3 to 5 show each library.

TradJS

Source: https://github.com/7flash/TradJS · `bun add tradjs`

Files are routes

File	Answers
`app/page.tsx`	`GET /`
`app/notes/page.tsx`	`GET /notes`
`app/notes/[id]/page.tsx`	`GET /notes/42` with `params.id = "42"`
`app/api/notes/route.ts`	`GET` / `POST /api/notes` (export functions named after methods)
`app/layout.tsx`	wraps every page below it
`app/middleware.ts`	runs before pages and routes below it
`app/not-found.tsx`, `app/error.tsx`	custom 404 and 500 pages
`public/robots.txt`	served as a file

Middleware hands typed values to pages

```
// app/middleware.ts
export default function middleware(req: Request, ctx: RouteContext<AppLocals>) {
  ctx.locals.user = readSession(req);
  if (new URL(req.url).pathname === "/notes" && !ctx.locals.user) return redirect("/login", 302);
}

// app/notes/page.tsx
export default function Notes({ locals }: PageProps<AppLocals>) {
  return <ul>{notesFor(locals.user!.id).map((n) => <li key={n.id}>{n.text}</li>)}</ul>;
}
```

Browser scripts mount where needed

```
// app/notes/page.client.tsx
export default function mount() {
  const timer = setInterval(refresh, 1000);
  return () => clearInterval(timer); // stopped when the page is hidden, restarted on return
}
```

Security that is on by default

- **Cross-site writes are refused.** A `POST` / `PUT` / `PATCH` / `DELETE` carrying cookies from another origin gets `403`.
- **Unsafe content never reaches the page.** `javascript:` / `vbscript:` / `data:` links, `srcdoc` and unsafe style URLs are dropped, on the server and in the browser, on first render and on updates.
- **Production errors reveal no details;** development errors show them.
- **The public folder is locked down:** no dotfiles and no path traversal in any encoding.

Static pages

`export const ssg = true` prerenders a page at startup, and `ssgParams()` lists the parameter values. A static page behind middleware still runs the middleware on each request, and is rendered with empty `locals`, so one visitor's data can never appear in the shared page.

Next: chapter 4, the data layer.

sqlite-zod-orm

Source: https://github.com/7flash/sqlite-zod-orm · `bun add sqlite-zod-orm`

One schema language

```
import { Database, z } from "sqlite-zod-orm";

export const db = new Database("data/app.db", {
  users: z.object({ email: z.string(), passwordHash: z.string() }),
  notes: z.object({ text: z.string().max(4000), userId: z.number() }),
}, {
  relations: { notes: { userId: "users" } },
  cascade: { users: ["notes"] },
  timestamps: true,
  softDeletes: true,
});
```

The tables are created from these schemas and migrated when they change. Columns are added, and renamed with the `renames` option. Changes that would lose or resurrect data are refused at startup with an explanation.

Typed reads, exact writes

You write	You get
`db.notes.select().where({ userId, text: { $like: "%milk%" } }).orderBy("id", "desc").paginate(1, 20)`	typed rows, a total count and the number of pages
`db.users.select().with("notes").all()`	each user with a typed `notes` array, loaded in one batched query
`db.notes.insert({ text, userId })`	the stored row, with `id`, `createdAt` and `updatedAt`
`note.text = "new"`	the change written to the database; the entity shows what was stored
`db.notes.update({ done: true }).where({ userId }).exec()`	the exact number of rows changed
`db.users.delete(id)`	a soft delete that cascades to notes; `restore(id)` brings both back

Hooks that respect transactions

`beforeInsert`, `beforeUpdate` and `beforeDelete` run inside the write and can refuse it. After-hooks run **once the outermost transaction commits**, and never for rolled-back work. So an email sent from `afterInsert` is never sent for a row that does not exist.

Live changes across processes

```
db.notes.on("insert", (note) => notify(note.userId));
```

Triggers log every write to a small journal, which keeps the last 10,000 changes. Every connection and every process on the same file sees every change, in order.

Next: chapter 5, logging.

measure-fn

Source: https://github.com/7flash/measure-fn · `bun add measure-fn`

```
import { createMeasure } from "measure-fn";
const api = createMeasure("api");

await api.root("GET /users", async () => {
  const users = await api("load users", () => db.users.select().all());
  return api("render", () => renderUsers(users));
});
```

```
[api] a → GET /users
[api] a-a → load users
[api] a-a ✓ 1.20ms → [2 items]
[api] a-b → render
[api] a-b ✓ 0.31ms → "<ul>…"
[api] a ✓ 1.74ms
```

Each line has a scope, a path showing nesting (`a-b` is the second child of `a`), a label, and on completion a status, a duration and a short result. The wrapped function returns exactly what it returned before. An error is logged and then rethrown unchanged.

Need	Use
synchronous code	`api.sync("label", () => ...)`
retries	`api.retry("label", { attempts: 3, retryIf }, fn)`
a fallback value instead of a throw	`{ catch: () => fallback }`
only failures and slow steps	`configure({ level: "errors", slowThreshold: 50 })`
nothing at all (tests)	`configure({ silent: true })`

Nesting survives `await` on Node.js and Bun through `AsyncLocalStorage`. URLs in logs have their secrets redacted.

Next: chapter 6, how all three are tested.

How we test

The three libraries have about 9,800 unit tests between them. They all passed, and the libraries still had real bugs. Each of those bugs only appeared when the code ran the way an app runs it:

- a real server talking to a real browser;
- a real database file written by an older version;
- default options rather than the convenient ones tests use.

So every guarantee in this book is checked by one of three kinds of test:

Kind	What it is	Example
Generated matrices	thousands of seeded cases compared with an independent answer	3,072 random queries, each compared with hand-written SQL on the same data; failures print the seed and the smallest failing case
Real-run tools	scripts that run the real product end to end	the starter app in headless Chrome (30 steps); an old published version builds a database file and the new version opens it
Measurements	numbers against a baseline, compared on the same machine	ORM speed as a fraction of hand-written `bun:sqlite`

Three rules make these tests worth trusting:

1. **A fix counts only if reverting it makes a test fail.** Every fix in the changelogs was checked this way, for example "66 of 71 cases fail without it".
2. **Look at the screenshots.** The browser tool saves one per step. One fix came only from looking at them: every note had an open edit box, so a page of 20 notes was a wall of text boxes.
3. **Measure the same way twice.** Machine load swung speed results by about 35% between runs, so speed is reported as a ratio to plain SQL measured in the same run. A change that did not measure faster was not released, even when it looked clever.



Fig. 1: a wrong password brings the login form back with a message (starter app, real-browser run)

Next: chapter 7, what this found.

What the tests found

All of these were present in released versions. All are fixed, each with a test that fails without the fix.

TradJS

Found by	Problem	Fixed in
browser run	a route returning plain text reached the browser as a download ("This site can't be reached") because adding a request id copied the response and lost its type	4.6.4
SVG check in Chrome	`<use xlinkHref="#icon">` drawn in the browser pointed nowhere, so sprite icons were blank	4.6.5
server vs browser matrix	`aria-hidden={true}` was written as an empty value on the server, so screen readers did not hide that content	4.6.3
renderer matrix	updates in the browser skipped the `javascript:` link check that the first render applied	4.6.2
request attack matrix	a plain-http page on the same host could post to the https site	4.6.1
production check	`tradjs start` from another folder made every page fail	next release

sqlite-zod-orm

Found by	Problem	Fixed in
query matrix	bulk `update().where()` / `delete().where()` with `{ $gte: 10, $lt: 20 }` used only the first operator and changed rows outside the range (present since 3.x)	4.0.8
an app using it on real data	a partial `update()` reset fields that were left out to their defaults	4.0.3
lifecycle matrix	`upsert` on a soft-deleted row revived it without its parent checks or hooks	4.0.7
change-feed matrix	with two processes on one file, the first to poll deleted the changes before the second saw them	4.0.10
upgrade tool	bulk writes reported twice the number of rows they changed	4.0.11
write-results matrix	after-hooks ran for writes that were then rolled back	4.2.0

measure-fn

Found by	Problem	Fixed in
behaviour matrix	turning on silent mode during a span dropped that span's end line	next release

Half of these would never show up in a demo. All of them could show up in production.

Next: chapter 8, speed.

Speed

An ORM costs time: validation, conversion, objects for every row. The question is how much. `bench/overhead.ts` in sqlite-zod-orm runs every operation twice on the same data, once through the library and once as hand-written `bun:sqlite` statements. It reports the library's speed as a percentage of the plain version, measured in the same run so that machine load cancels out.

Operation (10,000 rows, default options)	**4.2.0**	**4.2.2**
`get(id)`	12%	91%
`select().where()` range	9%	77%
`.with()` one level	10%	69%
`update(id)`	3%	58%
read and update one row (a game tick)	not measured	74%
change a field by assignment	2%	62%
`insert`	7%	35%
`upsert`	4%	10%

How the gains were made:

- Zod parsers and per-table plans are built once, not on every call.
- Rows are decoded only for the columns that need it.
- Statements per update went from three to two.
- SQL text and query plans are cached per shape.

One further round of changes did **not** measure faster: insert ×0.82 and upsert ×1.00, against run-to-run noise of about ±15%. It was not released.

> Note: `upsert` and `insert` are still the slowest paths. If your app inserts tens of thousands of rows per second, batch them with `insertMany` (61% of plain SQL) or write those statements directly with `db.raw`.

Next: chapter 9, limits.

Limits and what is next

What this stack does not do, and what is not yet proven:

Area	Status
Several machines sharing one database	not supported: SQLite is one file on one disk. Use one machine, or a database server with another library
Many processes on one machine	supported (WAL, write locks, live changes across processes); a load test at 1, 4 and 16 processes is in progress
React component libraries	not compatible: TradJS has its own small JSX renderer
`<feDropShadow>` in server-rendered SVG	the HTML parser predates this element, so the server's version parses as `fedropshadow`; it works when the browser renders it
measure-fn's cost per call	not yet measured cleanly; the next version (6.1) is released only after it is
The starter app (`tradjs init`)	passes 30 steps in a real browser; in review for the next TradJS release
Production build check	5 of its checks pass (hashed assets, no server code in browser bundles, byte-identical rebuilds, start from any folder); the rest are still to run

In progress:

- live updates over websockets in the starter app, instead of polling;
- a real-browser check of development mode (edits, error pages, recovery);
- a tool that runs every code example in the sqlite-zod-orm README;
- the multi-process load test.

Next: chapter 10, start an app.

Start in ten minutes

You need Bun 1.4 or newer.

```
mkdir my-app && cd my-app
bun init -y
bun add tradjs sqlite-zod-orm measure-fn
mkdir app
```

```
// app/page.tsx
import { db } from "../db";
export default function Home() {
  const notes = db.notes.select().orderBy("id", "desc").all();
  return <main>
    <h1>Notes</h1>
    <form method="post" action="/api/notes"><input name="text" required /><button>Add</button></form>
    <ul>{notes.map((n) => <li key={n.id}>{n.text}</li>)}</ul>
  </main>;
}
```

```
// db.ts
import { Database, z } from "sqlite-zod-orm";
export const db = new Database("app.db", { notes: z.object({ text: z.string() }) });
```

```
// app/api/notes/route.ts
import { db } from "../../../db";
export async function POST(req: Request) {
  db.notes.insert({ text: String((await req.formData()).get("text")) });
  return new Response(null, { status: 303, headers: { Location: "/" } });
}
```

Set JSX to TradJS in `tsconfig.json`:

```
{ "compilerOptions": { "jsx": "react-jsx", "jsxImportSource": "tradjs/client" } }
```

Run `bunx tradjs serve` and open the address it prints (http://localhost:3000, or the next free port if 3000 is taken). Add a note: it is stored in `app.db` and listed after the redirect. This exact app was run while writing this chapter.

From here:

- add `app/layout.tsx` for the page shell;
- add `app/middleware.ts` for sessions;
- add `page.client.tsx` where a page needs interaction.

The coming starter, `tradjs init --template app`, sets all of this up with accounts, sessions, editing, pages and live updates.

Source and documentation

- TradJS: https://github.com/7flash/TradJS
- sqlite-zod-orm: https://github.com/7flash/sqlite-zod-orm
- measure-fn: https://github.com/7flash/measure-fn

Compared with what you know

The next app has a page that reads notes, shows them in a list, accepts a note through a form, and redirects or refreshes the list after the write. It also needs schema changes without lost data and enough diagnostics to explain a slow request. The choice is not which library wins a generic benchmark. It is what code, runtime, deployment and operational obligations this particular app acquires.

This is a comparison of documented capabilities, not a shootout. What is said about TradJS, sqlite-zod-orm and measure-fn comes from the earlier chapters, where it was tested. What is said about the other tools comes from their official documentation and release notes, linked below, read in November 2026. We did not build the same app five times, count shipped bytes or rerun anyone's benchmark: where a cell says **not measured**, nobody here measured it. "Not documented" means the cited documentation does not promise that behaviour built in, not that it is impossible.

The page: choose a rendering framework

Framework	What you write for a form + list	JavaScript on a page with no interaction	CSRF for the form	Escaping and unsafe URLs	Deployment	Components and ecosystem	Better choice when
TradJS	`app/page.tsx` reads and renders; `<form action>` posts to `app/api/.../route.ts`; `page.client.tsx` only if needed.	None for a page needing no interaction; bytes **not measured**.	Rejects cross-origin cookie-bearing `POST` / `PUT` / `PATCH` / `DELETE` by default.	Text is escaped; dangerous link schemes, `srcdoc` and style URLs are removed on the server and in the browser (chapter 7).	Bun server, persistent local SQLite; prerendered routes via `ssg`. No edge or serverless target.	Its own JSX renderer; React components do **not** run unchanged. Ecosystem size **not measured**.	HTML navigation, ordinary forms, one durable host and small local scripts are sufficient.
Next.js App Router	Server Component `page.tsx` fetches the list; `<form action={serverAction}>` mutates it. Route Handler is another POST path. (F01)	Server Components add no *component* JS; `Link` uses client navigation. Total framework JS **not measured**. (F02, F03)	Origin/Host protection applies to **Server Actions**; do not assume it protects arbitrary Route Handlers. (F04)	React warns against raw HTML; Next warns that untrusted `router.push` URLs, including `javascript:`, execute code. (F06, F05)	Node server, Docker, constrained static export or platform adapters; platform-specific edge support varies. (F07)	React components and published shadcn/ui Next setup; library count **not measured**. (F25)	React component reuse, Server Components and platform integration matter more than having zero router runtime. (F02, F07)
Remix / React Router (Framework Mode, 7.12 or later; 8 is current)	Route `loader` returns list; route `action` processes `<Form method="post">`; loader data revalidates. (F08, F09)	With SSR, omit `<Scripts />` for a traditional HTML-only app; include it for hydrated routing. Bytes **not measured**. (F10)	**v7.12+** checks origins for UI action submissions by default; reverse-proxy host configuration matters. Not a blanket claim for arbitrary endpoints. (F11)	React raw-HTML escape hatch needs care; v7.12 fixed redirect validation and a server-rendered XSS case. (F06, F11)	SSR on Node/Cloudflare-style targets; static deployment for an appropriate build. (F12)	React libraries and documented shadcn/ui setup; ecosystem count **not measured**. (F25)	You want React, native-form progressive enhancement and automatic loader revalidation. (F09)
Astro	`.astro` page renders the list; an on-demand action or endpoint handles a normal HTML form. (F13)	HTML/CSS by default; client JS only for opted-in scripts/islands. Bytes **not measured**. (F14)	`security.checkOrigin: true` by default for specified unsafe form requests to **on-demand-rendered** pages. (F15)	Template text is escaped; `set:html` is not. A universal unsafe-URL denylist is **not verified** here. (F16)	Static build by default; SSR needs an adapter/server runtime. (F17)	Official React/Vue/Svelte/etc. integrations, with islands; ecosystem count **not measured**. (F18)	Content pages dominate and a few interactive islands or on-demand forms suffice. (F14, F13)

Framework	What you write for a form + list	JavaScript on a page with no interaction	CSRF for the form	Escaping and unsafe URLs	Deployment	Components and ecosystem	Better choice when
Hono + htmx	Hono `app.get` renders JSX/HTML; `app.post` returns a new HTML fragment; `hx-post` swaps it in. (F19, F24)	Plain Hono HTML sends no app JS; **htmx enhancement requires its browser script**. Bytes **not measured**. (F23)	Install `csrf()` middleware explicitly; not an automatic default for an unconfigured Hono app. (F20)	Hono `html` escapes ordinary interpolations, not `raw`; htmx requires escaping untrusted fragment content. Unsafe-URL policy **not verified**. (F21, F23)	Hono supports Bun, Node, Workers and serverless environments; persistence depends on host. (F22)	HTML/template-first; no automatic React-library compatibility. Extension count **not measured**. (F19, F23)	You want explicit HTTP handlers, partial HTML updates and runtime portability. (F22, F24)

Framework sources (official documentation and release notes):

- F01: Next.js App Router: a form can call a Server Action to mutate data.
- F02: Next.js: Server Components do not send component JavaScript; Client Components hydrate.
- F03: Next.js App Router client-side navigation and prefetching.
- F04: Next.js Server Actions compare Origin with Host to defend against CSRF.
- F05: Next.js warns untrusted router navigation URLs, including javascript: URLs, can execute code.
- F06: React rendering contract and danger of dangerouslySetInnerHTML with untrusted HTML.
- F07: Next.js deployments: Node.js, Docker, static export with limitations, adapters.
- F08: React Router framework route modules expose loaders, actions and components.
- F09: React Router Form uses browser form semantics, actions, progressive enhancement and automatic revalidation.
- F10: React Router 7: omitting Scripts while using SSR yields HTML-only navigation.
- F11: React Router v7.12.0 (2026-01-07) introduced UI action cross-origin CSRF checks and redirect validation; later release notes discuss checks.
- F12: React Router framework deployments include Node and Cloudflare templates and static hosting.
- F13: Astro form submissions can invoke actions without browser JavaScript when rendered on demand.
- F14: Astro default static HTML and CSS with JavaScript only on explicitly hydrated islands.
- F15: Astro default origin check on on-demand form requests; scope and CSP settings.
- F16: Astro escapes template expressions; set:html is unescaped and risks XSS.
- F17: Astro static output and SSR adapters.
- F18: Astro official integrations with React, Vue, Svelte and other frameworks.
- F19: Hono JSX SSR in route handlers; explicit JSX import source.
- F20: Hono optional CSRF middleware and protected form content types.
- F21: Hono html helper escapes interpolations but raw content requires manual care.
- F22: Hono supported runtimes including Bun, Node, Cloudflare Workers and serverless platforms.
- F23: htmx HTML attributes for AJAX fragments, progressive enhancement and need to escape untrusted injected HTML.
- F24: htmx hx-post causes asynchronous form POST and HTML swap.
- F25: Documented component library installation for React/Next.js, React Router and Astro; not an ecosystem count.

A "no interaction" page must not be confused with a *no-JavaScript configuration*. Astro and TradJS document HTML-first defaults; React Router 7 can render without `<Scripts />`; Next.js documents that Server Component code is not bundled for the client, **not** a zero-byte App Router shell. Hono can return static HTML without any browser script, but using `hx-post` entails htmx JavaScript. (F02, F10, F14, F23) None of these statements is a measured transfer-size or interactive-performance comparison.

The CSRF column describes **particular request paths**, not a security certification. A Server Action, a React Router UI action, an Astro on-demand form and a Hono handler do not necessarily share the same protection scope. Likewise, JSX text escaping does not prove that every `href`, navigation API, raw-HTML escape hatch or third-party extension is safe. The Next.js warning about `router.push` is a concrete counterexample to treating all frameworks' URL rules as interchangeable. (F04, F11, F15, F20, F05)

The database: choose a schema and persistence model

Data layer	Schema source	Migrations	Query types, relations and validation	Live changes across processes	Single file vs server database	Better choice when
sqlite-zod-orm	Zod object is table definition and input validator.	At startup: additive changes, declared renames; refuses destructive/resurrecting changes.	Typed queries, batched relations, soft deletes; hooks after outer commit.	Triggers/journal propagate writes to processes on the **same file**.	Bun + local SQLite file; multiple processes on **one host**, not many app machines. (D14)	One host, Zod already in use, and same-file change notifications remove extra infrastructure.
Prisma ORM v7	`schema.prisma` models + generated Prisma Client. (D01)	`prisma migrate dev` then reviewed/deployed migrations. (D02)	Generated typed CRUD and relations; form-input validation remains an application responsibility. (D01)	Equivalent SQLite cross-process journal **not documented** in cited core workflow; **not measured**. (D01)	Local SQLite via adapter, or PostgreSQL/MySQL/etc. over a connection; Bun SQLite adapter differs. (D01, D03)	A generated model client, explicit migration history or a server DB is valuable. (D01, D03)
Drizzle	TypeScript SQL table definitions. (D06)	`drizzle-kit generate` + `migrate`, or `push`; choose a migration workflow. (D07)	Inferred SQL-first typed queries and relations; Zod validation is separate unless added. (D06, D10)	Equivalent core same-file journal **not documented**; **not measured**. (D07)	Bun SQLite driver **or** remote SQL drivers (Postgres/MySQL/etc.). (D08, D09)	You want TypeScript schema plus control over SQL and migration files. (D06, D07)
Kysely	A TypeScript `Database` interface, often introspected/generated from an existing schema. (D11)	Code-authored migrations and migrator; not model-diff autogeneration. (D12)	Typed SQL query builder; runtime conversion/validation belongs to driver or app. (D11)	Equivalent core same-file journal **not documented**; **not measured**. (D11)	Driver/dialect for SQLite or server SQL databases. (D11)	SQL expressiveness, existing schema ownership and minimal model abstraction are priorities. (D11)
`bun:sqlite` **directly**	SQL `CREATE TABLE` and application-owned types. (D13)	Write/apply your own SQL; no ORM migration planner. (D13)	Prepared SQL; type/schema synchronization and validation are your responsibility. (D13)	SQLite offers concurrent file access, **not** an app change callback like sqlite-zod-orm's journal; write your own triggers/polling. (D13, D14)	In-process SQLite file on Bun; no server DB connector in this module. (D13)	You want full SQL control and can own schema evolution and event delivery. (D13)

Data-layer sources (official documentation):

- D01: Prisma ORM v7: schema.prisma, generated typed client, migrations, SQLite adapters and Bun-specific libsql guidance.
- D02: Prisma ORM v7 separate migration development and deployment commands.
- D03: Prisma ORM v7 database connectors including SQLite and server SQL databases.
- D04: Prisma ORM 8 release candidate status versus Prisma ORM 7.
- D05: Prisma ORM 8 release candidate database support, experimental SQLite.
- D06: Drizzle TypeScript table schemas and SQL-centric schema declaration.
- D07: Drizzle Kit migration generation, review, push and migrate choices.
- D08: Drizzle driver binding for Bun's SQLite file driver.
- D09: Drizzle connectivity across SQL databases and drivers.
- D10: Drizzle relational query definitions and querying relations.
- D11: Kysely TypeScript Database interface, query builder, runtime types determined by drivers and dialects.
- D12: Kysely authoring and executing migrations contrasted with generated model migrations.
- D13: bun:sqlite synchronous prepared statements, transactions, direct SQL and SQLite file handling.
- D14: SQLite WAL same-host shared memory, concurrent readers and single-writer characteristics.

The comparison is **Prisma ORM v7**, not an unqualified description of "Prisma." At this writing the official Prisma site identifies **v8 as a release candidate** with a changed contract/client/migration workflow; its SQLite package is **experimental**. Treat v7 as the generally available, well-documented reference in the table, and assess v8 separately before starting a new project. (D04, D05) A v7 SQLite-on-Bun project uses Prisma's documented adapter path, not `bun:sqlite` transparently. (D01)

A SQLite file is durable only if the host's storage is durable. WAL coordinates processes using the same host and does not work over a network file system. (D14) Choosing Next.js, React Router, Astro or Hono for distributed/edge deployment does **not** make the local file accessible from many ephemeral machines. If multiple app hosts require one shared writeable database, put a server database (or a specifically designed remote database

service) behind the app; both Prisma v7 and Drizzle document server-database connectors, and Kysely documents the relevant SQL dialects. (D03, D09, D11)

The **58–91%** in chapter 8 refers to selected *sqlite-zod-orm 4.2.2* operations against hand-written `bun:sqlite` SQL in the recorded same-machine benchmark. It does **not** describe all operations: the same run has `insert` at **35%**, `upsert` at **10%** and `insertMany` at **61%** of plain SQL. There is no measured Prisma/Drizzle/Kysely result in that run. These are *not measured* against one another; neither can a SQLite-only microbenchmark rank server databases. Use the existing benchmark as an overhead estimate for its stated operations, not as a cross-product league table.

The request: choose logs or traces

Logging / tracing	What you add to the handler	What is recorded by default	Nesting and timing	Output, retention and secrets	Runtime / deployment	Better choice when
measure-fn	`createMeasure("api")`; wrap request, DB call and render with `root(...)` / `api(...)`.	Per-step start/end lines, status, duration and short result; errors rethrown.	Nested paths through Bun/Node `AsyncLocalStorage`.	Console lines; Secrets in logged URLs are redacted. No OTLP export and no central retention.	Bun and Node; per-call cost **not measured**.	A single-process or same-host request needs an immediately readable duration breakdown.
pino	`pino()` logger; call `info` / `error`, add child bindings and explicit timers if needed. (L01)	Structured newline-delimited JSON log events, not automatic per-function span timings. (L01, L04)	Child fields group logs; **span hierarchy and elapsed times require more instrumentation**. (L01)	Configurable redaction and worker-thread transports to files/log services; storage/retention selected by operator. (L02, L03)	Node-focused logger; Bun/runtime overhead **not measured here**. (L01)	Searchable structured operational logs and configurable shipping matter more than step wrappers. (L02)
OpenTelemetry JS	Initialize SDK/exporter; manual `startActiveSpan` or auto-instrumentation libraries. (L05, L06)	Spans/attributes/context when instrumented; logs and metrics via separately configured SDK pipeline. (L05)	Parent-child spans, durations and cross-service context. (L05, L08)	Exporters/collector/backends configured separately; sensitive attributes need deliberate filtering. (L07, L09)	Documented Node and browser paths; Bun SDK compatibility and relative cost **not measured here**. (L10)	You need distributed traces, correlation across services and vendor-neutral telemetry. (L05, L07)

Logging and tracing sources (official documentation and project-maintained docs):

- L01: Pino JSON logger API, child loggers, configuration and structured fields.
- L02: Pino transports, destinations and worker-thread handling.
- L03: Pino configurable redaction for selected log fields.
- L04: Pino NDJSON default output and optional pretty transport.
- L05: OpenTelemetry JavaScript manual tracing with startActiveSpan and SDK configuration.
- L06: OpenTelemetry JavaScript instrumentation libraries and auto-instrumentation.
- L07: OpenTelemetry JavaScript exporters and span/log processing.
- L08: OpenTelemetry JavaScript async span/context propagation.
- L09: OpenTelemetry guidance and collector processors for sensitive data redaction.
- L10: OpenTelemetry Node.js SDK setup and exporter configuration.

These tools are not always substitutes. A duration wrapper answers “which function inside this request was slow?” A structured logger answers “what events and fields did this service emit?” Distributed tracing answers “where did a request go across services?” Pino may send log events through an OpenTelemetry transport, while OpenTelemetry can coexist with application logs. (L02, L07) No common workload was measured for their CPU cost, latency overhead or log volume. In particular, **do not** turn measure-fn's unmeasured cost into a “lighter than OpenTelemetry” claim.

Make the choice

Choose TradJS + sqlite-zod-orm when…

- Your next app is primarily server-rendered pages, ordinary links, forms and account-protected CRUD, with only a few islands of custom browser behavior. You accept TradJS's non-React JSX renderer and therefore do not need to import a React component library unchanged.

- Your data and WAL file can remain on one durable machine. Several processes may serve that host, but you are **not** planning a shared local SQLite file across serverless functions, edge regions or multiple application hosts. (D14)
- Your validation rules naturally fit Zod objects; creating and migrating SQLite tables from those definitions reduces the number of competing schema descriptions you maintain.
- You need transactional after-commit hooks or same-file cross-process change feed, and you are willing to operate and inspect the journal semantics yourself rather than assume a remote pub/sub bus exists.
- You need local nested timings while deciding whether to add broader logging and telemetry. You accept that measure-fn overhead and cross-tool log benchmarks are **not measured**.

Choose X instead when...

- **Choose Next.js App Router** when a React component system, Server Components, client navigation and hosting/platform integrations are significant requirements. Use Server Actions when you intend to rely on their documented CSRF check, and separately secure Route Handlers. (F01, F04, F07)
- **Choose React Router 7 Framework Mode (the Remix successor)** when React plus route loaders/actions, form revalidation and a deliberate no-JS SSR mode fit your team. Use **v7.12 or newer** for the documented origin-check fix, and verify proxy headers during deployment. (F08, F09, F10, F11)
- **Choose Astro** when the site is primarily content or statically generated pages with a limited number of on-demand forms and interactive islands; it also lets you reuse selected React/Vue/Svelte components. (F13, F14, F18)
- **Choose Hono + htmx** when you prefer writing HTTP handlers and swapping server-returned HTML fragments, including on an edge-capable runtime. Add CSRF middleware and escape fragment content as part of the implementation, not as optional follow-up work. (F20, F22, F23)
- **Choose Prisma v7** when you want its generated client, migration history and a supported path to a database service. Evaluate the changing **v8 release candidate** separately rather than assuming identical APIs. (D01, D02, D04)
- **Choose Drizzle** when TypeScript table definitions, SQL-like queries, reviewable migrations and Bun SQLite **or** PostgreSQL/MySQL connections are a better fit than Zod-defined tables. (D06, D07, D08, D09)
- **Choose Kysely** when an existing database is the source of truth and you want typed SQL with minimal ORM behavior; maintain or generate its `Database` interface as migrations evolve. (D11, D12)
- **Choose `bun:sqlite` directly** when your schema is small, queries are performance-sensitive and you are prepared to write SQL, migrations, runtime validation and change notification yourself. Chapter 8's direct-SQL baseline applies here, not as a guarantee of any production application's speed. (D13)
- **Choose pino** when machine-readable event logs, child bindings, redaction paths and configurable transports are immediate requirements. (L01, L02, L03)
- **Choose OpenTelemetry JS** when the user journey crosses services and you need trace context, sampling/exporters and backend correlation. Budget for instrumentation and sensitive-data policy; neither the integration work nor performance overhead has been measured in this chapter. (L05, L07, L09)

The decision is therefore architectural rather than fashionable: **a durable single-host HTML app** can use these three libraries without needing a client router, a server database or a tracing backend. Once the requirements explicitly demand React's components, multi-host persistence, edge-native storage or cross-service telemetry, use the corresponding alternative instead of forcing the local stack across a boundary it does not promise to cross.