FIREBACK
FRAMEWORK

Fireback is an AI-first, full-stack framework for building commercial software with the features and foundations that large-scale projects eventually need—available from the very first commit, at almost no additional cost or complexity.

Teams using Fireback typically ship around **10x faster**, while the same codebase can run on a **$5 server or scale to cloud, desktop, and mobile**. Every line of code belongs to your project, with no framework lock-in, and the entire stack can run on-premise.

Use it as a **single application or a set of services**. Fireback gives you a complete web interface, not just an API, so the common building blocks of a product are ready before you need them — not bolted on later.

Immediate Benefit

It is a foundation for cloud applications and backends in any industry, back-office tools, mobile apps — online or fully offline — and browser-only apps that need no server at all. Even a simple ticket-selling app needs far more than tickets: users and accounts, payments, administration, permissions, logs, auditing, notifications, and a back office. Fireback provides most of these building blocks ready to use, so a project ships from its first hour instead of months spent rebuilding the same foundations or gluing together commercial services and getting locked into their limits — a solid, tested foundation that cuts rewrite risk and multiplies team output.

Long-Term Benefits

The gains keep growing after the first project ships. What a team learns solving an edge case or extending a module is not spent on that one codebase — it goes back into the same modules every future project starts from, so knowledge compounds instead of resetting each time.

Teams working across several projects at once stop context-switching between different foundations: the same modules, conventions, and generated client code are already familiar wherever the work happens next.

Because the API is generated from the same definitions as the server, there is nothing left to hand-write as separate documentation — the contract *is* the documentation. And much of what ships is exactly the "nice to have" detail — audit logs, localization, permissions — that otherwise never gets built.

Technology Choices

Fireback uses **Go** as its backend foundation: fast, simple, portable, and compiled into a single binary, with proven libraries for routing, validation, databases, backups, and more.

The frontend is **not** tied to React — or to any frontend at all. Fireback can power React, Angular, Vue, desktop and mobile applications, IoT devices, or any client that can talk to its APIs. Its modules can also generate client code for different platforms, keeping clients in sync with the backend.

Fireback is designed to minimize the lock-in that often comes with frameworks. Modules are opt-in,

generated code is ordinary Go, and the runtime adds as little as possible. Where runtime logic is needed, it is plain Go code — the same kind of code a team would eventually write itself.

Fireback vs. emi

Fireback itself is a finished product, not a code generator: a set of Go packages and modules that work together, or individually, in an ordinary Go project. A team can ignore code generation entirely and write plain Go (or any other language) by hand, using only what Fireback ships. Generation itself is the job of a separate project, **emi** (§16) — Fireback's own modules simply happen to be written with it. Learning emi is not required, but it is strongly recommended: it is how Fireback's modules stay consistent with each other, it already covers a large share of what both small and large projects need, and understanding it before writing custom logic means not re-solving problems it has already solved.

How the Workflow Starts

Building on Fireback follows the same loop regardless of what the project ends up being — a back office, a consumer app, or a public API. The loop runs module by module, not feature by feature, and each pass through it produces a working, deployable increment rather than a half-finished branch.

1. **Scaffold the project.** A new project starts as a standard Go project — there is no bespoke project generator or wizard to run first. A team adds as many Fireback modules as the product needs (`abac` for users and permissions, `storage` for uploads, `finance` for payments, and so on), registering each one's `ModuleSetup` in `main.go` (§2). Nothing is pulled in that is not explicitly listed.

2. **Pick the client boilerplate.** Depending on what the project needs to ship, the same backend pairs with ready-made client boilerplate: the bundled React UI for a web app (§17), or starting points for Android and Swift/iOS clients. A team takes only the boilerplate its product actually needs, compiles it, and deploys — a real, running application, from day one, before a single project-specific feature has been written.

3. **Design the next feature in emi.** From there, new work is designed, not hand-coded, first: new endpoints, entities, and DTOs are declared in YAML module definitions (§16), then compiled through the emi compiler into Go, TypeScript, and the other generated artifacts a project has configured. Only once that shape exists does anyone write the business logic that fills it in — with an AI assistant, by hand, or a mix of both, since the generated scaffolding is ordinary Go either way.

4. **One definition, every surface.** The result of compiling a single feature definition is not just a REST endpoint. The same definition also becomes a CLI action, part of the WASM build (§3.14), an MCP server tool an AI agent can call, and generated client code for whichever languages a project targets — TypeScript, C++, C#, or others — picking only the targets a given project actually needs, never all of them by default.

5. **Repeat, module by module.** The cycle then repeats for the next module: design in emi, compile, write the business logic, ship. Because each pass produces a complete, working slice rather than a partial one, a project stays shippable throughout its own construction instead of only becoming shippable once everything is finished.

6. **Ship it, however it needs to run.** The same build deploys however the target actually requires: a Docker image for cloud and on-premise installs, a manual installer for teams that manage their own servers, or compiled libraries and app-store bundles for the mobile clients emi generated — any combination of these, side by side, rather than one deployment story forced on every project.

1 Features and Modules

The sections that follow walk through what Fireback actually does — users and permissions, file uploads, backups, notifications, payments, health monitoring, AI, and more — one feature at a time, in plain terms: what each one is and what it gives a project. The technical detail behind them comes later in this document, not here.

- Users, Roles & Workspaces (§1.1)
- Many Ways to Talk to the Same Backend (§1.2)
- Reactive Search (§1.3)
- Messaging & Notifications (§1.4)
- Backup & Restore (§1.5)
- File Uploads (§1.6)
- Running as a System Service (§1.7)
- Microservice or Monolith (§1.8)
- Interface Tools (§1.9)
- The Public Folder (§1.10)
- The React Dashboard (§1.11)

1.1 Users, Roles & Workspaces

Every project needs to know who is using it. Fireback lets people sign up, sign in, invite others to join, and change their own password. It also lets a project offer different types of workspaces — separate spaces for separate teams, customers, or tenants, each with its own members and its own settings.

Who can see or do what inside a workspace is controlled by assigning roles and access levels, rather than that logic being hard-coded into the application itself. All of this is handled by one module, called **abac**, written using Fireback the same way any other module is — it is not a special case. A microservice that has no need to manage its own users or workspaces simply does not include it.

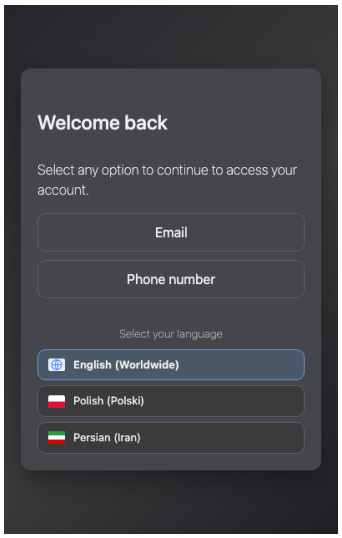

Figure 1: The sign-in screen, from the bundled UI.

Signing in is not limited to one method. People can log in with an email address, a phone number, Google, Apple, SSO, or Telegram — whichever combination a deployment chooses to turn on. Configuration can also vary by region: a project can offer different login methods in different parts of the world, and route email or text-message notifications through different providers per region, rather than being locked into one sender everywhere.

1.2 Many Ways to Talk to the Same Backend

Most backends only ever speak one language: an HTTP REST API, and nothing else. Fireback speaks several, from the exact same feature definitions.

Every action a backend exposes is automatically available as a REST endpoint, but also as a full CLI command — the same inputs become command-line flags, so nothing needs to be re-implemented or re-documented to reach a feature from a terminal or a script instead of an HTTP request. The same actions can also be exposed straight to an AI model or an MCP server as tools it can call, still governed by the same roles and permissions as everywhere else, rather than opening a separate, less-guarded door for AI access.

Input can arrive however a client finds it easiest to send: as JSON, as YAML, as multipart form data (file uploads included), or as a plain form postback — the same action accepts any of them, so a client does not have to conform to one narrow request shape. Real-time needs are covered the same way: Fireback's reactive actions add a live socket connection to a feature with very little extra work and no added risk, and the same mechanism can be reused anywhere in

a project rather than being wired up by hand each time.

Perhaps most unusual: Fireback can also run its own backend entirely inside a browser tab, compiled to WebAssembly, with a real Postgres database running alongside it in the same tab. That is what makes a rich, fully offline mode possible without writing and maintaining a second backend for it — the same server that runs on a real machine can also run disconnected, on a person's own device, and sync back up once a connection is available.

1.3 Reactive Search

Search is built to be reactive, not a single request-response round trip: results come back over the same live socket connection used elsewhere, updating as they arrive instead of waiting for one query to fully finish. A single search can also pull from more than one source at once, rather than being limited to one entity or one module.

Any module can define its own search handlers, contributing its own results to that same search without the modules needing to know about each other. The UI is not limited to a plain list of text results either: it can render whatever complex component best fits what came back — a user card, a document preview, a map — based on where each result came from.

1.4 Messaging & Notifications

Sending an email or a text message is never hard-wired to one provider. An admin UI lets a workspace configure which email and SMS providers to use, and a deployment can register several different senders at once — different providers, or different sending identities, for different kinds of messages or different regions — rather than every outbound message going through a single fixed account.

On top of that sits a full notification system, not just outbound mail: the same event can reach a person in app, as a mobile push notification, by email, or through whichever other channel a project has enabled, using per-workspace templates rather than one message hard-coded into the application.

1.5 Backup & Restore

A database is only as safe as its last backup, and Fireback treats that as a built-in feature rather than something every project has to set up for itself from scratch. Backups run continuously in the background, not as an occasional manual snapshot, and restoring is not limited to whatever moment a snapshot happened to be taken: a project can restore to any point in time within its retention window, down to close to the moment before something went wrong.

Underneath, this is real, production-grade backup technology, not a thin wrapper around a single dump command — the same kind of setup a dedicated database team would otherwise have to build and maintain by hand. It comes with its own history of what was backed up and verified, automatic health checks, and restore drills, so a project finds out a backup actually works long before the day it is actually needed.

1.6 File Uploads

File uploads are resumable by default: a large upload that gets interrupted partway — a dropped connection, a closed laptop lid — picks back up from where it left off instead of starting over, and the same applies to downloads. Where the bytes actually live is a deployment choice, not a hard-wired one: a project can store uploaded files directly in its database, or hand them off to S3-compatible object storage instead, without changing how the upload feature itself is used.

Every user can be given their own storage capacity, enforced automatically rather than left to grow unchecked. And uploads that are started but never actually attached to anything — abandoned partway, or never claimed by the feature that was supposed to use them — are swept up on their own, so storage does not slowly fill with orphaned files nobody remembers uploading.

1.7 Running as a System Service

A Fireback project installs itself as a proper background service on Windows, macOS, or Linux — started on boot, restarted if it crashes, managed the same way any other system service is, rather than needing someone to keep a terminal window open or babysit a process by hand.

Because a single static binary is all that runs, a project does not need Docker or a complex orchestration setup just to stay up. That matters most on the cheap end: several independent Fireback projects can run comfortably side by side on the smallest, cheapest 1GB VM a provider offers, rather than each one demanding its own container runtime

and the memory overhead that comes with it.

1.8 Microservice or Monolith

A Fireback project is never forced into one shape. The exact same modules can run as a single monolith, or be split apart and run as separate microservices, and a project can move between the two as it grows rather than committing to one architecture on day one.

That flexibility holds up once there is more than one instance behind a load balancer, too: real-time socket messages and event queues do not need to stay confined to whichever instance received them — they can be connected through a Redis pub/sub server, so a message published on one instance is still delivered to a user connected to a different one.

It also works the other way around: each module written for Fireback can live as its own separate project. Build ten modules once, and they are not stuck together — the same ten can be recombined into several different products, using only the modules each one actually needs.

1.9 Interface Tools

Two smaller features round out the UI itself. The first is the **app menu**: a searchable, hierarchical navigation menu that is driven from the backend rather than hard-coded into the frontend, with each item shown or hidden per user according to their permissions — so the menu a person sees always matches what they are actually allowed to do.

The second is **table view sizing**: every data table in the application can remember its own state per user — column widths, and whatever filters were applied — so a table looks and behaves the same way the next time someone opens it, instead of resetting to its defaults on every visit.

1.10 The Public Folder

A whole frontend project can be embedded directly into the final Go binary and served straight from it, rather than needing a separate web server or static host just to deliver the built site.

That still leaves room for search engines. Any given route can have its own page title, description, social preview tags, and structured JSON information spliced in as the page is served, without running a Node.js server just to render pages for that purpose. Since this is exactly what a crawler like Google's needs to index a page correctly, a project gets real, working SEO without taking on a separate, harder-to-operate rendering server to get it.

1.11 The React Dashboard

Fireback ships a set of React packages for building the frontend, and they are not a bare component library on their own: put together, they are a fully working, end-to-end dashboard, already exercised against a real backend, not a design mockup or a template that still needs its own logic wired in.

That dashboard is also a starting point, not a fixed product. The same packages used to build it can be pulled into a different project and reused there, giving that project a rich interface for the kind of complex, data-heavy screens — tables, forms, permissions, search — that normally take the most time to get right from scratch.

2 Architecture at a Glance

Fireback is distributed as a Go module, github.com/torabian/fireback, rather than a framework you install and configure from the outside. A project imports the modules it needs, calls each one's ModuleSetup in main.go, and gets that module's HTTP routes, CLI commands, and database migrations for free. There is no code-generation step baked into a "new project" wizard anymore — the recommended path is to clone the repository (or vendor it as a dependency) and continue from there.

Read this first: emi

A large share of what looks like "Fireback" behavior — entities, actions, DTOs, permissions, generated CRUD, the TypeScript SDK the React UI consumes — is not implemented in this repository at all. It is produced by **emi**, a separate compiler project (§16), from the *.emi.yml/*Module3.yml files this document keeps referring back to. Before spending much time in this codebase, it is worth reading emi's own documentation so its vocabulary (entities, actions, relations, the two parallel type systems in §16) is already familiar rather than being reverse-engineered from generated Go and TypeScript output:

https://github.com/torabian/emi/releases/latest/download/emi-docs.pdf

Core building blocks

- **Gin** — HTTP routing
- **urfave/cli v3** — the app / fireback CLI
- **tus** — resumable file upload protocol
- **pgx** — PostgreSQL driver (SQLite also supported since v1.6.0)
- **emi** — the external codegen compiler that turns module definitions into Go, TypeScript and OpenAPI

Configuration is intentionally narrow: a database connection is a single DB_VENDOR + DB_DSN pair (no more separate host / port / user / password / name keys), parsed and built by the small modules/fireback/dbdsn package shared between the setup wizard and the backup module. Everything else — gin mode, port, feature toggles — is read through the same app config <key> get/set CLI surface, so operators never need to hand-edit a config file to change one value.

The bundled React UI is a pre-built, embedded bundle shipped alongside the server binary, not a dev-server that hot-reloads against the Go source — a detail worth knowing before spending time editing UI source and wondering why nothing changes.

2.1 Three ways to run the same core

The exact same framework described throughout this document backs three different deployables, not just one server binary:

- **The real microservice** (cmd/fireback) — the standard case this document mostly describes: modules registered in main.go, a real TLS-capable HTTP server, a real database connection, the full CLI surface.
- **The WASM build** (cmd/fireback-wasm, §3.14) — the same module system and the same gorm-based entity logic, compiled to WebAssembly and run entirely client-side against an in-browser Postgres, with no network backend at all. This is what powers the wasm-demo frontend build (§17.3).
- **A canary preview instance** (§3.10) — the ordinary server binary, but connected through a deliberately read-only database role, with migrations refused outright, for safely previewing a build against production-like data without risking a write.

None of these are separate codebases: they are the same application.Application/ModuleProvider machinery (§3.1) assembled by three different entry points, which is precisely what makes the module system's decoupling (§3.2) load-bearing rather than cosmetic — a module written against the shared abstractions works in all three without modification.

3 The Core Framework

Everything else in this document sits on top of modules/fireback — the package that defines what a "module" is, how a request becomes an authorized action, how configuration and errors are shaped, and the shared field types every entity uses. This section is a tour of that foundation.

3.1 The module system

The core abstraction is application.ModuleProvider (modules/fireback/application/Application.go) — a plain struct, not an interface, that a module's XModuleSetup(cfg) *ModuleProvider constructor populates and returns. A provider can carry HTTP wiring (GinWebServerInitHooks), CLI commands (CliHandlers, plus AppendCli for handlers needing the live *Application), entity migrations bundled per feature (EntityBundles, each with permissions, auto-migration entities, CLI commands, mock providers), a goose migration directory for plain modules (GoMigrateDirectory *embed.FS), permission declarations, a config-struct accessor (ConfigProvider), a startup hook (OnAppStart), translations, and seeders. Builder methods (ProvideCliHandlers, ProvidePermissionHandler, ProvideConfigHandler, ProvideEntityHandlers, ...) let a module append to the struct incrementally. fireback's own module is the minimal case:

```
func FirebackModuleSetup(setup *FirebackModuleConfig)
  *application.ModuleProvider {
  module := &application.ModuleProvider{
    Name: "fireback",
    GoMigrateDirectory: &migrations.MigrationsFs,
  }
  module.ProvideCliHandlers(firebackModuleCliHandlers())
  module.AppendCli = firebackModuleAppendCli()
  return module
}
```

"Opt-in" is literal: nothing runs unless a project's main.go puts a module's *ModuleProvider into application.Application.

Modules. cmd/fireback/main.go builds that slice by calling each module's setup function directly — fireback.FirebackModuleSetup(nil), backup.ModuleSetup(nil), storage.StorageModuleSetup(nil), wallet.WalletModuleSetup(&WalletModuleConfig{...}), eventbus.ModuleSetup(nil), and so on — and assigns the slice to xapp.Modules. A module never self-registers through an init(); a feature absent from this list contributes zero routes, CLI commands, migrations, or config keys to the binary. internalstats.ModuleSetup in main.go illustrates the decoupling this buys: internalstats never imports abac, but main.go wires its Authorize callback to call resolve.ResolveActionContext with an abac SecurityModel, so one project's binary can bind an otherwise-independent module to whichever auth engine it chose, without a compile-time dependency between the two.

Once Modules is assembled, fireback.CommonHeadlessAppStart(xapp, onDbReady) (Entrypoint.go) drives startup: load env config, call every module's ConfigProvider() once, initialize logging, open the DB pool, invoke each module's OnAppStart(x) hook (used by e.g. eventbus to start a goroutine only when it is actually registered), and hand off to RunApp(x).

3.2 The action/context pipeline

fireback core supplies only the vocabulary this pipeline is built from: Security.go defines a base SecurityModel (AllowOnRoot, ActionRequires, ResolveStrategy), QueryDSL.go defines the request-context struct every action consumes, and Error.go defines IError. The actual authorization *engine* lives in modules/abac/resolve, which declares its own structurally identical SecurityModel and the resolution machinery described fully in §9.6. The shape, in outline:

1. resolve.ResolveActionContext(request, securityModel) is the single entrypoint an emi-generated action handler calls, regardless of transport — it inspects request.GetGinCtx()/GetCliCtx() to pick a concrete resolver (HTTP header vs. CLI-configured workspace/token).
2. resolve.WithAuthorizationPureDefault(context) resolves the bearer token to a user, loads that user's per-workspace access, and checks it against the security model — refusing outright if AllowOnRoot is set and the caller isn't in the root workspace.
3. For plain gin middleware (not an emi action), resolve.AuthorizeRequest/WithAuthorizationFn wrap the same pair and abort the request directly on failure.

The Authorize callback pattern in main.go's internalstats.ModuleSetup call is the idiom end to end: the module's own signature only knows about a generic request context and returns a fireback.QueryDSL; the closure supplied by main.go is what actually calls into abac's resolver and translates the result — letting a module accept *any* auth engine a project chooses, without depending on one at compile time.

3.3 Configuration

Configuration.go defines the core Config struct with envconfig/description tags on every field (e.g. DbDsn string `envconfig:"DB_DSN" ...`), package-level defaults, and LoadConfiguration(), which overlays .env/ environment values. DB_DSN is deliberately the *only* persisted database setting — there is no parallel DB_HOST/DB_PORT/DB_USERNAME/DB_PASSWORD/DB_NAME; anything needing the individual pieces goes through modules/fireback/dbdsn.

dbdsn is a small, deliberately dependency-free package (it does not import the rest of fireback) with a ConnectionInfo{Host, Port, Username, Password, Database, SSL} struct and vendor-dispatched Parse/Build functions for postgres and mysql/mariadb. BuildPostgres has a deliberate quirk: an empty keyword is omitted entirely rather than written as password=, because pgx.ParseConfig silently drops the rest of the DSN after an empty password= token:

```
func BuildPostgres(info ConnectionInfo) string {
  parts := []string{}
  if info.Host != "" {
    parts = append(parts, "host="+quotePostgresValue(info.
        Host))
  }
  if info.Password != "" {
    parts = append(parts, "password="+quotePostgresValue(info
        .Password))
  }
  // ... other keys, same pattern
  sslmode := "disable"
  if info.SSL { sslmode = "require" }
  parts = append(parts, "sslmode="+sslmode)
  return strings.Join(parts, " ")
}
```

The app config <key> get/set CLI surface has two layers. ConfigurationCli.go is itself emi-generated: for every config field it produces a cli.Command named after the field's kebab-case name with get/set subcommands, ending in config.Save(".env"). CliActions.go assembles the top-level config command, prepending hand-written db/dbdsn/ssl/sqllog wizard subcommands ahead of the generated per-field ones. On top of that, ConfigRegistry.go provides a *cross-module* listing: GetCombinedConfigInfo reflects over fireback's own Config plus every registered module's ConfigProvider() result, using only the envconfig/description tags — so any module's emi config: block is folded automatically into fireback config list with no per-module registration step. Values that look like secrets (matched against SECRET/PASSWORD/TOKEN/PRIVATE/DSN in the env var name) are masked as ****** unless –reveal-secrets is passed.

3.4 Core data types

All defined under modules/fireback/complexes, each implementing driver.Valuer/sql.Scanner plus gorm's GormDataType/GormDBDataType/GormValue interfaces so they behave as first-class column types across sqlite/mysql/mariadb/postgres:

- TString — map[string]string, a locale → text map (e.g. {"en": "Hello", "fa": "..."}), serialized to one JSON/JSONB column. Accepts a bare string (wrapped into the default locale), a {locale: value} object, from JSON, YAML, or plain text alike. Get(locale) falls back to the default locale, then to any present value, then "". Used pervasively for translatable Name/Description fields.
- MJson — a generic opaque JSON blob column, with MJsonFrom/MJsonTo[T] converters and SQL-builder helpers for cross-dialect JSON path queries/updates.
- XDateTime / XDate — string-backed date(time) types parsed via a permissive date parser, normalized to RFC3339, with special-cased handling for pre-1500 years (treated as a Jalali/Iranian calendar date) and computed metadata helpers ("days left," "in range," locale-formatted display).
- complexes.Time — a nullable-time wrapper with a Present bool flag distinguishing "field absent" from "field explicitly null" during JSON/YAML decoding; this is the role PlainTime plays in entity definitions — nullable and presence-aware, but not locale-aware like XDate.
- Also present: TMoney/money field types, a duration type, and XFile, the file-reference field type used by storage-backed entities (§10).

3.5 CLI framework

fireback runs entirely on urfave/cli/v3. RunApp(xapp) (RunAppCli.go) builds one root cli.Command with shell completion enabled, a global –al flag (accept-language, default en-us), and Commands: GetCliCommands(xapp). GetCliCommands recursively walks every module's CliHandlers, the result of module.AppendCli(x) when set, and every entity bundle's CliCommands, then appends fireback's own built-ins (init, env, tasks, doctor, service, config, start, seeders, public-folders, about, version).
The cliShort convention — visible both in emi .yml definitions and in generated Go — is a short mnemonic alias for an action's CLI subcommand, wired in as cmd.Aliases = []string{meta.CliShort} in the generated *ActionCli.go file. This is what lets, for example, fireback appmenu g <id> work identically to fireback appmenu get <id>.

3.6 What a generated entity file looks like

Real per-entity *.dyno.go files are build artifacts, not checked into this repository — but the vendored github.com/torabian/fireback dependency ships a complete example (modules/payment/PaymentConfigEntity.dyno.go) that illustrates the shape:

- A ...Qs "queryable fields" struct (one QueriableField per column, tagged with CLI flag name, table name, type, and query-string key) backing the CLI's –query filter DSL.
- The entity struct itself, always carrying fireback's common envelope fields ahead of its own: Visibility, WorkspaceId, LinkerId, ParentId, IsDeletable/IsUpdatable, UserId, Rank, an internal auto-increment ID plus a public UniqueId, CreatedAt/UpdatedAt/ DeletedAt, and computed *Formatted display fields — before its module-specific columns.
- An actions-dispatch struct (e.g. paymentConfigActionsSig) of function pointers assigned to a

package-level var, so hand-written custom logic can call through a stable indirection point rather than the free functions directly.

- CRUD implementation functions (...ActionCreateFn, ...ActionUpdateFn, ...ActionGetOneFn, ...QueryFn, ...), plus seeding, batch-insert, in-memory caching, page-at-a-time streaming, and export/import helpers, all generated once per entity.

The *action*-level generated shape actually used by this repository today (illustrated by modules/abac/interfacetools/defs/AppMenuGetAction.go and its ...Cli.go sibling) generates, per action: a metadata function (name, CLI name/short, URL, method, description), typed request/response structs, an IsGin()/IsCli() transport-detection pair, an HTTP handler builder, a gin handler builder, a CLI handler builder, and typed Go client call helpers — one source of truth reaching HTTP, gin, CLI, and a typed client simultaneously.

3.7 Pagination and query conventions

fireback.QueryDSL is the shared request-context struct threaded through every action: StartIndex (offset), Cursor (a separate cursor-based path), ItemsPerPage, Sort, Query (a filter-DSL string), SelectableColumn (projection), Deep (whether to join nested arrays/objects), and WithPreloads. CommonCliQueryDSLBuilder is the CLI-side constructor (–query/–offset/–cursor/ –limit, capped at 1000); CommonQueryFlags is the shared flag slice every entity's list command reuses (–verbose, –x-select, –minimal, –sort, –deep, ...). Every list action returns results alongside a uniform QueryResultMeta{TotalItems, TotalAvailableItems, Cursor} envelope. Hand-written/raw-SQL queries against embedded .sql/.vsql templates go through UnsafeQuerySqlFromFs/ContextAwareVSqlOperation, which run a counter query (a ...Counter.sql sibling supplies TotalItems) alongside the main query, substituting @limit/@offset/@internalCondition placeholders — including a sqlite-only sentinel where limit = -1 means "no limit," since sqlite treats a negative LIMIT that way while postgres/mysql simply omit the clause.

3.8 Error handling

ferror.Error is the canonical, dependency-free error family: {Message ErrorItem, MessageParams, MessageTranslated, Errors []*FieldError, HttpCode}, where ErrorItem is itself a locale → text map mirroring TString's shape — so every error already carries every translation, and ToPublicEndUser(translatable) projects it down to one language's public error for the actual response, hiding internal-only detail. fireback.Error.go re-exports this family as type aliases (IError = ferror.Error, ...).

Construction helpers cover the common cases: Create401Error and friends for auth failures, CastToIError for normalizing any Go error into the envelope, and GormErrorToIError for the DB-specific case — gorm.ErrRecordNotFound becomes a translated 404, everything else a 400, with the raw native error text included only when the app isn't running in production. Every module gets its own localized message catalog generated the same way: string constants plus a singleton whose fields are ErrorItem maps keyed by locale (en/fa/pl, plus a machine-readable "$" code key), so callers can compare err.Is("ResourceNotFound") without string-matching human-readable text. The package doc on ferror frames it candidly as "a standalone copy... matching the original (since-drifted) intent," and floats — not yet adopted — the idea of wrapping it on github.com/rotisserie/eris later for stack traces, rather than pretending the current shape is final.

3.9 Validation

Validation.go's CommonStructValidatorPointer[T] drives go-playground/validator/v10 over a DTO — the same validate: struct tags emi's tags: { validate: required } convention (§16) compiles down to — and turns each field error into an IErrorItem{Location, ErrorParam, Message, Type} ready to append onto an IError's Errors slice. One detail matters for anyone writing a PATCH-style update action: the helper deliberately skips required tag failures on patch/update calls, since a partial update request is expected to omit fields it isn't changing — only a full create enforces required in the strict sense.

3.10 Canary deployments and load balancing

Two smaller subpackages support running fireback behind a blue/green-style rollout: canary/ builds and inspects a *read-only* Postgres role for a canary preview instance (via database/sql with the pgx driver rather than gorm, one of the narrow spots pgx is used directly, §3.13) — and, as already

noted in §3.13, ApplyMigration refuses to run at all when CanaryMode is set, since a canary connects through a role that structurally cannot write. loadbalancer/ provides the counterpart multi-instance routing support. Both are wired in through ordinary CLI command groups (CanaryCli.go, LoadBalancerCli.go) rather than being always-on background behavior.

3.11 CLI wizards and environment discovery

clitools/ (excluded from wasm builds) holds the interactive setup experience: DatabaseWizard.go walks an operator through picking a vendor and test-connecting a candidate DSN before it's saved (this is the second of the two narrow, direct pgx usages — test-connecting during the wizard doesn't go through gorm at all, since there's nothing to migrate or query yet), InitWizard.go drives first-time project setup, and SSLCli.go handles the manual/self-signed/Let's-Encrypt (autocert) certificate choices consumed by SSL.go. These wizards register themselves into CliActions.go's command tree through package-level init() function-pointer variables, rather than being called directly — the one place in the module system where an init()-based registration pattern is used, and only because these commands need to exist in the non-wasm build without clitools itself becoming a hard compile-time dependency of CliActions.go. envm/ is the lower-level counterpart: .env file discovery and loading (LoadFirebackAppConfiguration, EnvironmentUris) that both the wizards and LoadConfiguration itself build on.

3.12 The filter DSL

The Query string on fireback.QueryDSL (§3.7) isn't just passed through to SQL verbatim. jsonsql/ and queryexpr/ implement a small JSON-logic-to-SQL translator: a structured filter expression (the same shape a frontend form-builder or saved-search feature would naturally produce as JSON) is walked and compiled into a parameterized WHERE clause fragment, rather than every entity's generated QueryFn hand-rolling its own ad hoc filter parsing. This is also what ContextAwareVSqlOperation's @internalCondition placeholder is filled with when a hand-written .vsql template needs to splice in whatever filter the caller supplied.

3.13 Database layer: gorm and pgx

gorm is the ORM used for all entity persistence. DirectConnectToDb(config) is the single connection-construction path: it picks a gorm dialector from DB_VENDOR (gorm.io/driver/mysql, gorm.io/driver/postgres, or a vendored cgo-free sqlite dialector chosen deliberately over gorm.io/driver/sqlite for portability), applies a table-prefix naming strategy, and returns *gorm.DB. For sqlite specifically the pool is forced to a single connection, since concurrent sqlite connections were observed to freeze on some environments. Every complexes.* field type implements gorm's data-type interfaces specifically so raw Create/Find/UpdateColumns calls handle them transparently across dialects.

jackc/pgx/v5 is present and used, but not as the primary ORM driver — it appears in narrow, targeted spots: creating and inspecting a read-only Postgres role for canary preview instances, test-connecting to a candidate DSN during the interactive setup wizard, and (see below) as the low-level protocol the wasm build's database bridge has to imitate. The split is deliberate: gorm for all ordinary entity CRUD, pgx for direct, low-level Postgres operations that need role/session control outside gorm's abstraction.

Migrations split by module shape. Entity modules are migrated via plain dbref.AutoMigrate (bundle.AutoMigrationEntities...), called from ApplyMigration(xapp, level) for every module's entity bundles. Plain (non-entity) modules instead ship SQL migrations under a migrations/ directory with a //go:embed *.sql index.go, consumed via goose: RunMigrationBasedOnGoose sets a per-module goose table name and calls goose.Up. ApplyMigration runs goose against every module's GoMigrateDirectory when set (this is how fireback's own migrations get applied) and AutoMigrate for entity bundles — a single module can mix both mechanisms if it has both. It refuses to run at all in canary mode, since a canary preview instance connects through a deliberately read-only DB role.

3.14 The WASM build path

cmd/fireback-wasm/main.go (build tag wasm) is a separate binary entry point that never calls CommonHeadlessAppStart/RunApp — the CLI-dispatch path is entirely excluded from wasm builds,

since it transitively pulls in things with no wasm platform support (interactive readline prompts, socket listeners, OS signals). Instead it builds the same application.Application struct directly, connects through application.ConnectWasmPostgres, and dispatches requests from a JS bridge call straight into an HTTP mux, with no real socket involved.

The trick that makes the same gorm-based core "just work" in-browser: a small driver wraps a JS callback (backed by **pglite**, a WASM-compiled Postgres running in the same tab) as an ordinary *sql.DB, handed to gorm's postgres dialector with PreferSimpleProtocol: true (pglite has no server-side prepared-statement cache to reuse across "connections") and a distinct driver name (so gorm's migrator doesn't inject a pgx-only sentinel query argument the wasm driver can't marshal to a JS value). Everything above the dialector — gorm.Open, AutoMigrate, Find, Create, transactions, hooks, and every complexes.* field type — is unaware it isn't talking to a real TCP connection. A migration-skip cache table (deliberately not itself an AutoMigrate-managed entity) lets a repeat page load within an hour skip re-running AutoMigrate's column-introspection round trips, which are otherwise a visible load-time cost over the JS bridge. This is the same core that §2 onward describes for the server binary — the wasm build is a different *entry point*, not a different framework.

Selected files in modules/fireback

- FirebackModule.go / .dyno.go — module definition + generated permission/error-code catalog.
- Entrypoint.go — CommonHeadlessAppStart, the generic non-wasm bootstrap sequence.
- CliActions.go / RunAppCli.go / ConfigurationCli.go / ConfigRegistryCli.go — CLI command trees, per-field config get/set, combined config listing.
- Configuration.go / ConfigRegistry.go — the Config struct and cross-module config reflection.
- DatabaseConnection.go — gorm dialector setup per vendor.
- MigrationManager.go — goose + AutoMigrate orchestration.
- QueryDSL.go / Query.go / QueryResultMeta.go / CrudCoreActions.go — the request-context struct and raw/templated SQL execution helpers.
- Error.go / ferror/Error.go — the IError family.
- Validation.go — validator integration.
- Security.go — base SecurityModel (superseded operationally by modules/abac/resolve's own copy).
- GinRouting.go — wires every module's GinWebServerInitHooks.
- HttpServer.go / SSL.go — TLS/ACME-listening HTTP server (excluded under wasm).
- application/ — Application, ModuleProvider, PermissionInfo; the wasm connection path.
- complexes/ — TString, MJson, XDateTime, XDate, Time, money/duration/ file field types.
- dbdsn/ — per-vendor DSN parse/build, no fireback dependency.
- wasmpgdriver/ — the JS-bridge database/sql driver backing the wasm build's Postgres connection.
- sqlitedriver/ — the cgo-free sqlite dialector.
- migrations/ — fireback's own embedded goose migrations.
- ferror/ — the standalone error-envelope package.
- envm/ — .env discovery/loading.
- clitools/ — interactive setup wizards (database, init, SSL), !wasm-only.
- gintools/ — gin middleware and static-embed serving helpers.
- canary/, loadbalancer/ — canary-deployment read-only DB role support and multi-instance routing.

- jsonsql/, queryexpr/ — the JSON-logic-to-SQL filter-DSL translator behind QueryDSL.Query.

4 Technology Choices

Every dependency in Fireback was picked on purpose, not accumulated. The guiding rule is the same on both sides of the stack: reach for a well-known, widely used library where one already does the job well, and only write something in-house where nothing suitable exists.

4.1 Backend: Go

The server is written in **Go**. For a project that has to run reliably for years, in the hands of teams who did not necessarily write the original code, Go's own trade-offs are exactly the ones that matter most: a small, simple language that is fast to learn and hard to misuse, compiled to a single static binary with no runtime or dependency tree to install on a server, and fast enough — both to run and to build — that "compile and deploy" stays a non-event instead of a chore. Concurrency, memory safety, and a strong standard library come built in, rather than needing a separate ecosystem of add-ons to reach production quality.

Well-known libraries, not reinvented ones

- **Gin** — HTTP routing and middleware
- **go-playground/validator** — struct and field validation
- **pgx** — PostgreSQL driver
- **go-sql-driver/mysql** — MySQL/MariaDB driver
- **wal-g** — battle-tested backup/restore, the same engine large-scale Postgres operators already trust
- **urfave/cli** — the app / fireback command-line surface

Every one of these is a mature, widely deployed project with its own community, its own track record, and its own maintainers — not a bespoke wrapper that a team would have to understand, debug, and keep alive by itself. On top of that foundation, Fireback adds a small number of its own modular libraries only where nothing off-the-shelf covered the need well enough — resumable file uploads (built on the tus protocol) being the clearest example: a real gap, filled once, and reused by every project instead of being solved again per codebase.

4.2 Frontend: modular React packages

The bundled UI is built the same way, in **React**, but it is not shipped as one monolithic app a project either takes whole or leaves alone. It is split into roughly sixteen independently versioned @fireback/* packages — authentication screens, admin/back-office screens, the resumable-upload widgets, the wallet UI, and so on — each installable on its own into any React project, whether or not the rest of Fireback is in use. A team that only needs a ready-made login flow, or only the file-upload widget, can take just that package.

4.3 Client code, already generated

Because every entity, action, and permission is already declared once in emi's module definitions (§16), Fireback does not stop at generating a TypeScript client for its own bundled UI. The same definitions are emi's compilation targets for the client code a wider range of platforms and languages need to talk to the same backend — so integrating a new client is, in the vast majority of cases, a matter of generating that code rather than hand-writing and hand-maintaining it for every platform a product ships on.

5 Users & Workspaces

Identity in Fireback is organized around **workspaces**: every user belongs to one or more workspaces, and almost every permission check is scoped to "this user, in this workspace." A special **root** workspace type exists for platform operators, distinct from ordinary tenant workspaces created by customers.

5.1 Signup and passports

Authentication is pluggable through *passport methods* — email, and other identity providers can be enabled or disabled per deployment. PassportMethodSyncSeeders is what seeds the default set (e.g. enabling "email") on a fresh database, which is why make devsetup runs ./app seeders before creating a root user.

5.2 Members, roles and invites

Workspace membership supports:

- Browsing and removing members (root and workspace-admin capable)
- Changing a member's role within a workspace
- Public join links — a workspace can expose a slug-based **public join key**, letting a classic signup join that workspace automatically via a shareable link instead of an explicit invite

> *A blank granted capability is treated by* MeetsCheck *as matching everything — so a bare workspace member with no capabilities explicitly denied can end up with de facto full access. Always grant narrow, explicit capabilities rather than leaving them blank.*

5.3 CLI shortcuts

```
./app auth --in-root=true --value=a@a.com \
  --type=email --password=123321 \
  --workspace-type-id=root \
  --first-name=Root --last-name=User
./app ws view
./app user mock --count=5
```

The auth command above (full name authorize) is explicitly marked CLI-only in its own source — "never expose in HTTP" — because flagged, non-interactive mode (CreateAdminTransaction) can create a root account directly, bypassing every normal signup gate. Run with no flags at all, it instead drives an interactive wizard: it discovers which passport methods are currently enabled by calling the same check the classic sign-in flow uses (§9.1), walks whatever Next steps come back (OTP, password creation, sign-in), and on success writes the resulting token and workspace straight into .env — which is exactly what make devsetup (§2) automates non-interactively for local development.

6 The Login Flow

The screens below are not mockups: they're captured straight from the real **manage** UI by e2e/cypress/e2e/login.cy.ts as part of that spec's normal run, and dropped into docs/latex/figures/login.cy.ts/login/ as fixed, overwritten-every-run PNGs (see screenshotsFolder in e2e/cypress.config.js — Cypress nests named cy.screenshot() output under a subfolder per spec file, so the path includes login.cy.ts even though the screenshot names themselves are just login/*). Since this doc build reads them from that same path, running the spec again after a UI change and rebuilding the docs is enough to keep these figures current — there's no separate image to hand-capture or remember to update. The spec captures both palettes in one run (toggling body's dark-theme class directly), and each figure below picks whichever one — light or dark — matches the palette of the PDF currently being built (\iffirebackdark, see main-body.tex).

6.1 Landing on the welcome screen

Visiting /manage/#/welcome against a server with at least one passport method enabled shows the "Welcome back" screen: whichever passport methods (email, phone, ...) the deployment has enabled are offered as sign-in/sign-up options.

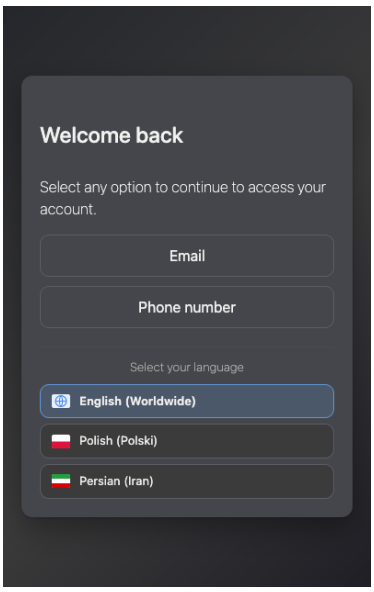

Figure 2: The welcome screen, testWelcomeBackWhenMethodsEnabled.

6.2 Continuing with email

Picking the email method (#using-email) routes to /selfservice/email, where the user enters the address they want to sign in or register with. The same field drives both sign-in and account creation — Fireback branches to "Complete your account" only if that email doesn't already have a passport on record.

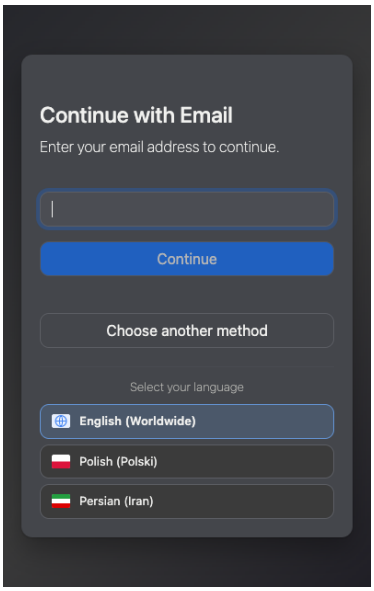

Figure 3: The email step, inside testCreateAccount.

> *These two figures are generated artifacts, not committed screenshots someone updated by hand — login.cy.ts regenerates them as a side effect of asserting the same UI states it already tests, so a real UI change that breaks the flow fails the spec before it can ship a stale picture into this document.*

7 ABAC Permissions

Permissions live in modules/abac, Fireback's attribute-based access control layer. Every protected action checks the caller's **capabilities** within the current workspace before running.

7.1 Capability keys

Capability keys are hand-written string literals per file — there is no shared NewCrudPermissionSet helper generating them. New code should grep sibling files for the naming convention already in use (e.g. a root. prefix historically, later stripped from root.manage.* / root.modules.* except root.modules.* for ROOT_ALL_MODULES, kept distinct from the wildcard ROOT_ALL_ACCESS ("*")) rather than invent a new naming scheme.

Common abac primitives

- CapabilitiesTree — lists the full capability tree for a role/workspace
- ChangeWorkspaceMemberRole — promotes/demotes a member
- AcceptInvite / AddUserToWorkspace
- CheckClassicPassport — email/password login flow
- Bot actions — service-account style access without a human passport

7.2 How other modules plug in

Modules outside abac (internalstats, backup, storage, ...) never import abac directly. Instead they accept an Authorize callback in their own ModuleConfig; a project's main.go wires abac's root-only or capability check into that callback explicitly. Left nil, the default authorizer still requires a root-workspace token via fireback.ResolveActionContext — it only becomes a real enforcement point once some auth provider assigns fireback.AuthorizeRequest, which is exactly what abac.WorkspaceModuleSetup does.

This indirection is deliberate: it keeps low-level modules decoupled from the specific authorization model a given product wants, while still giving every module a consistent authorization seam to hook into.

7.3 What "attribute-based" means in practice

A capability check in this codebase is never just "does this role have permission X." Every check is implicitly scoped by two attributes attached to the caller's current request: which *workspace* they are acting in, and which *user* they are. The same person can hold different capabilities in different workspaces simultaneously (through separate workspaceRole grants, §8), and a capability granted at the workspace-type level (via a workspaceType's default role) is distinct from one granted directly to an individual member's role — both are checked, and both must be satisfied for an action gated with AllowOnRoot: false plus a non-empty permission set. This two-attribute scoping (user × workspace, rather than a flat global role) is what the "A" in ABAC buys over a simpler RBAC model: the same role name can carry different real permissions depending on which workspace it was granted in, without needing a separate role per workspace.

7.4 Reading a permission failure

When a check fails, the caller gets back a translated IError (§3.8) rather than a bare HTTP status — typically AbacMessages.NotEnoughPermission

for a capability mismatch, or a token-specific message (TokenNotFound, an expired-token sentinel) when the resolution failed earlier in the pipeline, before a capability was even checked. Distinguishing "your token doesn't work" from "your token works but lacks this permission" matters operationally — the former usually means a client needs to re-authenticate, the latter means an administrator needs to grant a capability — and the pipeline in §9.6 is structured so those two failure modes never collapse into the same generic 403.

8 ABAC Entity Reference

modules/abac/Abac.emi.yml defines twenty-two entities. Every one of them carries the same four housekeeping fields unless noted otherwise: workspaceId (which workspace the row belongs to), userId (who owns/created it), and createdAt/updatedAt (PlainTime). The entries below describe what is specific to each entity — its purpose, its own fields, and how it relates to the others.

Entities at a glance

capability	builtin/custom permission catalog
passportMethod	enabled login methods (email/phone/oauth)
pendingWorkspaceInvite	invite awaiting a not-yet-created user
preference	per-user settings (timezone, ...)
userProfile	per-workspace display name override
publicJoinKey	shareable "join this workspace" link
emailConfirmation	pending email verification code
phoneConfirmation	pending phone verification code
token	long-lived credential (sessions, bots)
workspaceInvite	active invite to a known email/phone
userWorkspace	membership: user × workspace
workspaceRole	role grant: userWorkspace × role
bot	service account (role, no passport)
publicAuthentication	in-flight signup/OTP state machine
notificationConfig	per-workspace email/sms templates
workspaceConfig	root-only global auth toggles
role	named bundle of capability keys
workspaceType	workspace kind → default role
user	the person (identity, not credentials)
workspace	the tenant/organization
passport	one login credential for a user
notification	one in-app message to one user

8.1 capability

The catalog of everything that can be granted, builtin or custom (cliShort: cap). Only two real fields: name and description, both TString (locale → text maps, e.g. {"en": "Query user", "fa": "..."}). A capability's own UniqueId/Name/Description mirror application.PermissionInfo, seeded by CapabilityUpsertPermissionFn — this is the table §7's hand-written CompleteKey string literals get registered into, and what CapabilitiesTree walks to render a role-editing UI.

8.2 passportMethod

Which login methods are switched on, per region (cliShort: method). type is an enum — email/phone/google/facebook — and region defaults to (and in open-source Fireback is pinned to) global. clientKey holds the OAuth client id for provider methods like Google. This is the row PassportMethodSyncSeeders (§5) writes to enable "email" on a fresh database.

8.3 pendingWorkspaceInvite

An invite sent to a value (email or phone, typed by the sibling type field) that has *no user account yet* — distinct from workspaceInvite below, which targets a specific known email/phone with richer metadata. Carries coverLetter, workspaceName (a denormalized display copy), and an optional roleId granted on acceptance.

8.4 preference

One row of per-user settings, currently just timezone. Scoped by both userId and workspaceId, so the same person can, in principle, carry different preferences per workspace.

8.5 userProfile

A per-*workspace* override of how a user is displayed — firstName/lastName here are distinct from the same fields on the user entity itself: user is the global identity, userProfile is what that identity looks like inside one particular workspace (useful when the same person is known by a different name/title per tenant).

8.6 publicJoinKey

Backs the **public join key feature**: a slug-based link anyone can use to join a workspace without an explicit per-person invite. Just a roleId (the role granted to whoever joins) plus the standard workspace/user/timestamps fields — the slug/lookup mechanics live in PublicJoinKeyLookup, consumed by the classic-signup join path.

8.7 emailConfirmation / phoneConfirmation

Twin entities, one per channel: a status, the email or phoneNumber being verified, a random key (the code/token sent to the user), and expiresAt. Short-lived by design — these exist only for the window between "we sent a code" and "the code was confirmed or expired."

8.8 token

A bearer credential: userId it belongs to, the opaque token string itself, and validUntil (XDateTime). Used for ordinary session tokens *and* reused by bot (§8, below) as the record a bot's tokenId points at — a bot's token is refreshed by writing a new token row and repointing tokenId, never mutating the old one in place, so a revoked/expired token stays intact for audit.

8.9 workspaceInvite

The richer, targeted counterpart to pending WorkspaceInvite (cliShort: invite): a publicKey hash letting an admin hand someone a direct confirm/signup link without an email actually being sent, a coverLetter, targetUserLocale for translating the invite UI, explicit email/phonenumber, required firstName/lastName, a required roleId granted on acceptance, and forceEmailAddress/forcePhoneNumber flags that lock the invitee out of changing the contact method an admin already chose for them.

8.10 userWorkspace

The membership join row — *the* entity that says "this user belongs to this workspace," unique on (workspaceId, userId) (gormMap enforces a composite unique index userworkspace_idx). Three fields (userPermissions, rolePermission, workspacePermissions) are tagged gorm: "-" / sql: "-" — computed at request time from the member's workspaceRole rows and never persisted, so a permission change takes effect immediately without a migration or cache-bust.

8.11 workspaceRole

The actual role *grant*: links one userWorkspace to one role, unique on (userWorkspaceId, roleId) (workspacerole_idx). Splitting membership (userWorkspace) from role grants (workspaceRole) into two entities — rather than one row with a role column — is what lets a single member hold multiple roles in the same workspace simultaneously.

8.12 bot

A service account: a role and a long-lived token, no human passport (§7 *"Common abac primitives" — Bot actions*). roleId is required up front; creating a bot then materializes three more rows behind the scenes, each one referenced back by id: botUserId (a user with no passport attached), userWorkspaceId (that user's membership), and workspaceRoleId (the grant of roleId to that membership). tokenDuration is an enum — 90/180/indefinite days — and tokenId points at the bot's current token row, replaced wholesale (not edited) whenever the bot's credential is rotated.

8.13 publicAuthentication

The scratch space for an in-progress signup/OTP flow (cliShort: pa) — everything that has to survive between one HTTP request and the next before an account fully exists. totpSecret/totpLink hold a freshly generated 2FA secret and its QR-code URL when TOTP is required at signup; sessionSecret is a long hash that re-authenticates the caller after they confirm an OTP, so the rest of the signup form can be filled in without re-proving identity; passportId/passportValue tie the row to the credential being created; blockedUntil is a Unix timestamp lockout after too many OTP attempts. recoveryAbsoluteUrl is tagged sql: "-" — computed for the response, never stored.

8.14 notificationConfig

Per-workspace templates and defaults for outbound messaging (cliShort: config): which email/SMS provider id (generalEmailProviderId/generalGsmProviderId) and sender id to use for each of three message families — workspace invites, forgot-password, and confirm-email — each family carrying a Content/ContentExcerpt pair the workspace can override, plus a read-only Default/DefaultExcerpt pair (tagged sql: "-", computed from the application's built-in templates, never persisted) shown as a fallback/reset target in the UI. cascadeToSubWorkspaces and forcedCascadeEmailProvider let a parent workspace's config apply to its children instead of each sub-workspace configuring its own.

8.15 workspaceConfig

The one genuinely global entity: root-workspace-only, effectively a singleton (*"at the moment, a single record is allowed, and only for root workspace"*) holding the auth-flow toggles that apply to the whole deployment — enableRecaptcha2, enableOtp and its stricter siblings requireOtpOnSignup/requireOtpOnSignin, enableTotp/forceTotp, and forcePasswordOnPhone/forcePersonNameOnPhone for phone-based signup. The doc comment is explicit that sub-

projects should never repurpose this entity for product-specific settings — add a new config entity instead.

8.16 role

A named, reusable bundle of capability keys (see §7). name is a TString; capabilitiesListId is an MJson field holding the list of granted completeKey strings directly as JSON. The doc comment explains why: this *replaces* an older many-to-many role_capabilities join table, because Emi has no relation mechanism compatible with Fireback's string-uniqueId foreign-key convention (the xId fields every other entity here uses). isDeletable/isUpdatable (both default true) let a small set of built-in roles be marked protected.

8.17 workspaceType

The template a workspace is created from (cliShort: type): a title (TString), a URL-safe slug (must start with /, lowercase a–z and dashes only, ≤50 chars — enforced in ValidateTheWorkspaceTypeEntity, with gorm: unique as the DB-level backstop), and a required roleId. That role must already exist and belong to the *root* workspace — it is what actually defines a workspace of this type's default capabilities, e.g. separate student / teacher / school workspace types in a multi-tenant education product.

8.18 user

The person, independent of how they log in — credentials live on passport, not here. Required firstName/lastName, optional photo/avatar/gender/title/birthDate (XDate), lastIpAddress, an embedded primaryAddress object (street/city/state/postal/country), and free-text phoneNumber/jobTitle/company/bio. The description notes management of this entity directly is *root only* — ordinary workspace members see themselves through userWorkspace/userProfile, not raw user CRUD.

8.19 workspace

The tenant/organization itself (cliShort: ws): a name (TString), required typeId pointing at the workspaceType that governs it, and an optional parentId — workspaces can nest into a tree, which is what lets notificationConfig's cascadeToSubWorkspaces and the root/sub-workspace distinction elsewhere in this module make sense.

8.20 passport

One login credential for one user (§5 — "Signup and passports"): type (matching a passportMethod), a unique value (the email address, phone number, or OAuth subject id), and a password field explicitly excluded from both JSON and YAML serialization (json: "-", yaml: "-") so it can never leak into an API response or an exported dump. thirdPartyVerifier pins an OAuth-created passport to its provider so a passwordless login can't be attempted through a different channel; totpSecret/totpConfirmed hold this specific credential's own 2FA state, separate from any account-wide TOTP policy in workspaceConfig. A single user can hold several passport rows — e.g. both an email and a phone credential for the same identity.

8.21 notification

One in-app message to one specific recipient (userId, required): title, body, and an isRead flag. The doc comment is explicit that these are not created through the generic per-entity create endpoint — a root-only SendNotification action creates them in bulk, one row per recipient, so a notification always has a real, resolvable recipient and (when sent by a person rather than the system) a real senderId.

> *Reading order for a new relation:* workspaceType *names a* role*; a* workspace *picks a* workspaceType*; a* user *joins a* workspace *via* userWorkspace*; that membership is granted zero or more roles via* workspaceRole*; and a* role*'s actual permissions are just a JSON list of* capability *keys on* role.capabilitiesListId*. No entity in this module points at another through an emi* type: one / collection *relation — every cross-reference here is a plain* string? xId *field resolved by hand in Go, per the* role *entity's own comment about Emi's relation mechanism not fitting Fireback's string-uniqueId convention.*

8.22 Schema diagram

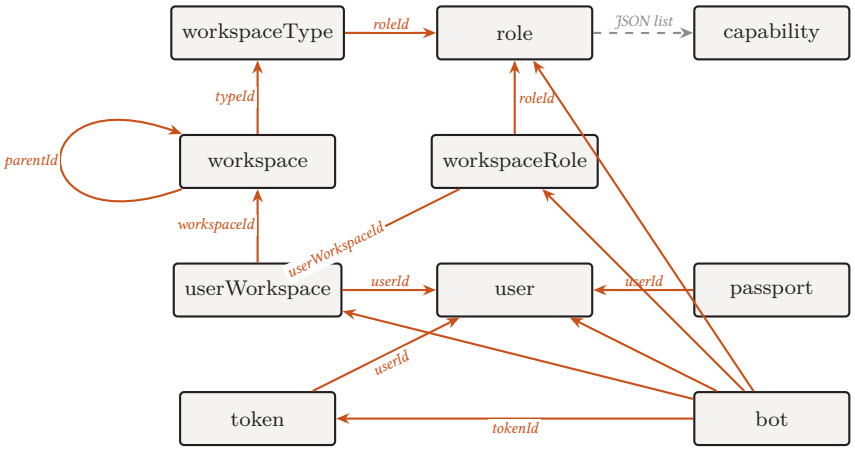

Fig. 1 — ABAC core schema. Solid red arrows are plain *Id string references resolved by hand in Go, not emi type: one/collection relations (§8). The dashed arrow is role.capabilitiesListId, a JSON array of capability keys stored directly on the row — not a foreign key at all. bot is the busiest entity in the module: creating one writes a user, a userWorkspace, a workspaceRole and a token row and keeps an id pointer to each of the four, on top of the roleId it was given up front. workspace's parentId loop is how workspaces nest into a tree (§5). Twelve more entities hang directly off user and/or workspace by that same userId/ workspaceId convention, with no further chain, and are left off the diagram to keep it legible: preference, userProfile, notification, emailConfirmation, phoneConfirmation, publicAuthentication, notificationConfig, workspaceConfig, workspaceInvite, pendingWorkspaceInvite, publicJoinKey, and passportMethod — each described in §8.

9 ABAC Implementation Notes

The entity reference above (§8) describes *what* modules/abac stores. This section covers *how* the hand-written Go logic layered on top of those entities actually behaves — the signup/login state machine, capability checking (including a real, shipped bypass), workspace membership, bot lifecycle, token handling, and how other modules hook into all of it.

9.1 Signup and login flows

Every public (unauthenticated) action in abac goes through resolve.ResolvePublicActionContext — unlike resolve.ResolveActionContext, it performs no token lookup and no permission check; it just extracts Workspace-Id from the request and hands back an AuthResultDto shell. The convention throughout the package is that the HTTP/action wrapper (CheckClassicPassportAction, ClassicSigninAction, ClassicSignupAction, ...) is a thin shim, and the reusable logic lives in a lower-case *Core function (checkClassicPassportCore, classicSigninCore, classicSignupCore, ...). This is what lets AuthFlow.go — the CLI-only auth command — reuse the exact same business logic end-to-end without going through Gin at all.

Discovery. CheckClassicPassportAction validates a raw value's shape (email regex, phone regex, or an anonymous_* sentinel), loads WorkspaceConfigEntity for the root workspace, and looks for an existing PassportEntity. It returns a Next array (e.g. ["otp"], ["signin-with-password"]) computed from RequireOtpOnSignin/EnableOtp and whether the passport has a password or TOTP secret — if OTP is the only viable next step, it implicitly requests one on the caller's behalf.

Sign-in. ClassicSigninAction re-checks the same recaptcha/OTP-forced config, validates the password with security.CheckPasswordHash, and layers TOTP on top: if ForceTotp is set and the passport has no *confirmed* TOTP secret, a fresh secret is minted and the response instructs Next: ["setup-totp"] instead of completing sign-in; if a *confirmed* secret already exists and EnableTotp is on, the caller's TotpCode is required and validated. Only then does applyUserTokenAndWorkspacesToToken run, minting/reusing a session token via AuthorizeWithToken and, over HTTP, setting a secure authorization cookie.

Signup. ClassicSignupAction defers to completeClassicSignupProcess, which builds fresh User/Role/Workspace/Passport structs and calls

the central UnsafeGenerateUser with createUser/ createWorkspace/createRole/ createPassport all true. That function wraps everything in one gorm transaction: passport rows are always stamped to the *root* workspace, then the user, then CreateWorkspaceAndAssignUser (workspace + userWorkspace row), then the workspaceRole grant, then a session token. If ForceTotp is on, the response short-circuits with ContinueToTotp: true instead of a completed session.

publicAuthentication as scratch state

PublicAuthenticationEntity is the single glue row threading state between requests during signup/OTP/TOTP/password-reset: SessionSecret proves continuity across a "check → otp → signup" sequence; TotpSecret/ TotpLink hold a pre-generated TOTP secret *before* an account exists; IsInCreationProcess distinguishes "this OTP is for a brand-new account" from "this OTP is a login/reset for an existing one." Rows are deleted once their flow completes, so a value cannot be replayed.

OTP mechanics. Requesting an OTP deletes any prior publicAuthentication row for the same value, generates a 6-digit code (fireback.GenerateRandomKey(6)), stamps a configurable lockout window, and sets IsInCreationProcess: true when no matching passport exists yet. Redeeming an OTP looks up the newest matching row by (PassportValue, Otp), checks the lockout hasn't passed, and either reports ContinueWithCreation: true (handing back SessionSecret/TotpUrl) or, for an existing passport, mints a session token and deletes the row.

9.2 Capability checking — and a real bypass

The atomic check is resolve.MeetsCheck:

```
func MeetsCheck(required []PermissionInfo, perms []string) bool
    {
  meets := true
  for _, req := range required {
    hasExact := slices.Contains(perms, req.CompleteKey)
    hasParental := false
    for _, a := range perms {
      if strings.Contains(req.CompleteKey,
        strings.ReplaceAll(a, "*", "")) {
        hasParental = true
      }
    }
    if !hasExact && !hasParental { meets = false }
  }
  return meets
}
```

The wildcard match (stripping * from a granted key, then checking it's a substring of the required key) is what lets a granted "abac.workspace.*" satisfy a required "abac.workspace.update". It is also exactly what produces the **blank-capability bypass**: if a granted capability string a is "", stripping * from it is still "", and strings.Contains(anything, "") is unconditionally true in Go. A single blank entry anywhere in a caller's resolved capability list makes every required permission pass. This is not hypothetical: it arises whenever a workspaceRole row has no roleId, or a workspaceType has no roleId configured — the capability-aggregation pipeline then produces an empty-string capability rather than no entry at all, so a bare workspace member can end up with de facto full access.

> *The gap cannot be closed inside* MeetsCheck *itself without breaking roughly 180 existing tests that rely on it to let root act on typeless test workspaces as a bare member. Instead, membership-mutation actions add an independent, stricter gate:* requireRealWorkspaceMembersCapability *re-derives the caller's actual granted capabilities directly from* workspaceRole/ role *rows, explicitly skips blank entries, and requires a real exact or prefix match. It is a no-op for root, which keeps trusting the wildcard the same way every other root-only action does.*

MeetsAccessLevel is the higher-level gate: it short-circuits true for the literal wildcard ROOT_ALL_ACCESS = "*" held by *both* the user and the workspace, then calls MeetsCheck twice — once against role-derived capabilities, once against workspace-type-derived capabilities — requiring both to pass. (A code comment records a previously-shipped regression where this function always returned true regardless of the computed verdict, silently letting every capability-gated action through; it now genuinely enforces both checks.)

CapabilitiesTree powers the admin role-editing UI: it loads the full capability catalog, computes the caller's own capabilities, and filters the catalog to only what the caller could actually grant — so a role editor never lets you hand out a capability broader than your own — before assembling a terminal.Tree keyed on dotted capability strings for RolePermissionTree.tsx (§17) to render.

Every hand-written permission set in the package (PERM_ROOT_BOT, PERM_ROOT_TOKEN,

PERM_ROOT_NOTIFICATION, ...) is a literal permission literal of the shape

```
PermissionInfo{
  CompleteKey: "abac.<kebab-name>.<action>", ...
}
```

— there is no shared helper generating them, deliberately preserving the exact string shape older generated code produced so already-granted role rows in a live database keep resolving. CapabilityUpsertPermissionFn is what actually registers each declared PermissionInfo into the capability table during SyncPermissionsInDatabase, only inserting a row when the lookup returns a genuine gorm.ErrRecordNotFound (an earlier version treated *any* lookup error as "missing" and could crash the app on a duplicate-key insert).

9.3 Workspace membership

Four distinct shapes exist, split along "root acting on an arbitrary workspace" vs. "a workspace admin acting on their own current workspace":

- AddUserToWorkspaceAction / RemoveUserFromWorkspaceAction / ChangeUserWorkspaceRoleAction — root-only, take an explicit workspaceId body field since the caller's own header is pinned to root.
- ChangeWorkspaceMemberRoleAction / RemoveWorkspaceMemberAction — self-service, always operate on auth.WorkspaceId (never a body field), gated by both the shared permission *and* requireRealWorkspaceMembersCapability. ChangeWorkspaceMemberRoleAction fully *replaces* a member's role assignments rather than appending; RemoveWorkspaceMember refuses to let a caller remove their own membership, preventing an admin from locking themselves out.
- BrowseWorkspaceMembersAction — lists membership rows scoped to the caller's own workspace, enriched per-row with name and role (documented as "no batching," acceptable since membership lists are small).

AcceptInviteAction resolves the caller, looks up the workspaceInvite by its unique id, and inside one transaction creates the userWorkspace row for the *caller* (not the invite's own userId, which is only an audit field for who *created* the invite — a documented bug fix), creates the matching workspaceRole grant, and deletes the invite so it cannot be redeemed twice.

The public join key feature layers on top of ClassicSignupAction: a signup request carries either workspaceTypeId (create a new workspace, the default path) or publicJoinKeyId (join an *existing* workspace/role via JoinExistingWorkspaceAndAssignUser). A separate, unauthenticated PublicJoinKeyLookupAction resolves a (workspaceSlug, joinKeySlug) pair to a join-key id and display info, 404ing without revealing which half of the pair failed.

9.4 Bot / service accounts

Bots are ordinary workspace members with no passport, authenticated purely by a long-lived token row — GetUserFromToken has no notion of "session" vs. "bot" token at all. BotCreateAction is self-service (gated on the caller's *current* workspace, not root-only) and creates rows in a fixed order: a bare user (no passport), a userWorkspace membership, a workspaceRole grant, a fresh token (GenerateSecureToken(32), expiry per tokenDuration — 90/180 days, or empty for indefinite), and finally the bot row itself, storing only the linkage: botUserId, userWorkspaceId, workspaceRoleId, tokenId. The plaintext token is returned exactly once in the create/refresh response and again on a direct get — by design, since token rows are stored unhashed and there is no other way to view a token again once the response has scrolled by.

Rotating a bot's credential (RefreshBotTokenAction) deletes the *current* token row and creates a brand-new one, repointing bot.tokenId — never mutating the old row in place, so a revoked/expired token stays intact for audit. Deleting a bot cascades in reverse creation order (role grant, membership, token, user row, then the bot row), each step best-effort so a partial prior delete doesn't block cleanup.

9.5 Token lifecycle

One token entity is shared by session and bot tokens; nothing in the schema distinguishes them beyond how they were created and their validUntil. AuthorizeWithToken mints/reuses session tokens: it supports a "jwt" strategy (HS256, 1-hour exp claim) or the default plain random 32-byte token, and — before minting a new one — reuses the caller's first still-valid existing token rather than piling up rows on every sign-in. A fresh session to-

ken's validUntil is stamped 30 days out (a code comment records this used to be an inert placeholder ignored entirely by validation, until expiry enforcement was added for bot tokens).

Validation on every incoming request is centralized in resolve.GetUserFromToken: a lookup by token string, then — if validUntil is non-empty — a comparison against the current time. An expired token returns the same sentinel error as "no such token," so the caller can't distinguish the two cases.

9.6 How other modules plug in

WorkspaceModuleSetup() is the function registered in Fireback's combined module list: it wires migrations, the abac config: block, Gin init hooks, every permission set via ProvidePermissionHandler, and the full CLI command tree. Contrary to an older "Authorize callback" model, authorization no longer goes through a generic injection point at all — every module that needs it calls modules/abac/resolve directly: resolve.ResolveActionContext(c, &resolve.SecurityModel{...}), which internally chains WithAuthorizationPureDefault → GetUserFromToken + GetUserAccessLevels + MeetsAccessLevel. A separate resolve.AuthorizeRequest Gin-middleware adapter exists for routes not modeled as emi actions — chiefly WebSocket subscriptions, which read Workspace-id/ Role-id/ Authorization from query params (a handshake can't carry custom headers) and must report auth failures over the socket frame itself, since the HTTP writer is no longer usable once the connection has been hijacked.

Root-only vs. workspace-scoped is encoded structurally through SecurityModel{ActionRequires, AllowOnRoot}:

```
if security.AllowOnRoot &&
    context.WorkspaceId != fireback.ROOT_VAR {
     return nil, &fireback.IError{
       HttpCode: 400,
       Message: AbacMessages.ActionOnlyInRoot,
     }
}
```

CapabilityCreateAction is AllowOnRoot: true despite being CRUD-shaped, specifically because the capability entity has no workspaceId/userId column at all — it is a single global catalog every workspace's role editor renders from, so only root may extend it.

9.7 Notifications

SendNotificationAction is root-only: given a list of user ids plus a title/body, it creates one notification row per recipient (skipping, rather than failing the whole batch on, any id that doesn't resolve). Template rendering lives in a small messaging subpackage: rows keyed by (region, key, locale) are rendered as Go text/template strings against a substitution map, falling back to the "en" definition of a catch-all region: "any" row — and finally to an in-binary default template — if nothing more specific exists, on the principle that it is better to send an English message than an error. Provider dispatch for both email and SMS switches on a type field on the configured provider row (sendgrid, mailgun, postmark, resend, smtp, terminal for local dev, ...), decoding each provider's own JSON config into a small typed struct before making the actual call.

9.8 Access-level resolution internals

GetWorkspaceAndUserAccesses is the function that actually produces the UserHas/WorkspaceHas capability lists MeetsAccessLevel (§9.2) checks against: it walks the caller's userWorkspace row for the active workspace, follows every attached workspaceRole to its role, and flattens each role's capabilitiesListId JSON array into one string slice — this is the exact point where a workspaceRole row with no roleId (or a workspaceType with no roleId configured) degrades into the blank-string entry behind the bypass discussed in §9.2, rather than simply contributing nothing. A companion function, ResolveWorkspaceTypeRoleCapabilities (WorkspaceCoreFeatures.go), runs the same projection the other direction at *workspace-creation* time: given a workspaceType, it resolves the default role that type points at and the capabilities that role carries, which is what a freshly created workspace's first member inherits before anyone has explicitly granted them anything.

GetSqlContext/QueryDSL (resolve/GetSqlContext.go, resolve/QueryDSL.go) are the last link in the chain: once ResolveActionContext has produced an AuthResultDto, these translate it into the fireback.QueryDSL an entity's generated Browse/Get call actually receives — stamping WorkspaceId/ UserId from the resolved auth context rather than trusting anything the caller supplied in the request body, which is what makes "every protected action is scoped to this user, in this workspace" (§7) true at

the data-access layer and not just at the permission-check layer.

The CLI's ws scope command (GetUserAccessScop WorkspaceCli.go) is a thin, read-only wrapper around this same resolution path — useful for an operator debugging "why doesn't this role see this button" without needing to reproduce the failing HTTP request.

9.9 CLI surface

- auth (alias authorize) — the flagship interactive or flagged end-to-end auth flow; explicitly marked CLI-only (*never expose in HTTP*) since flagged mode can create root accounts non-interactively.
- ws (alias workspace) — the largest surface: CRUD for workspace/type/config/invite/role/membership join-key, plus scope (dump a user's resolved access), as (set the CLI's active token/workspace), view (run the authorization check against the current CLI credentials and print the result), and a misc group including a totp simulator.
- user — CRUD for user/token, plus accept-invite and user mock (hand-written, not emi-generated — seeds *N* realistic mock users through the same code path the real create endpoint uses, for local dev/demo data).
- bot — CRUD plus refresh-bot-token.
- role — CRUD for roles.
- abac config — per-field get/set for this module's own config block, namespaced separately from the top-level app config surface.
- abac internal — CRUD for lower-level entities: public-join-key, public-authentication, preference, user-profile, pending-workspace-invite, and capability (including capabilities-tree).
- parse — decodes any token (JWT or internal) and prints the workspaces/access associated with its user, useful for debugging a captured token.

Selected files in modules/abac

- AbacModule.go — module registration: permissions, migrations, config, CLI trees.
- resolve/*.go — the auth/authz core: token validation, capability aggregation, MeetsCheck/MeetsAccessLevel, Gin and CLI adapters around one shared implementation.
- AuthFlow.go — CLI-only interactive/flagged auth flow.
- ClassicSigninActionImplementation.go, ClassicSignupActionImplementation.go, ClassicPassportRequestOtpActionImplementation.go, ClassicPassportOtpActionImplementation.go, CheckClassicPassportActionImplementation.go, ConfirmClassicPassportTotpActionImplementation.go — the login/signup/OTP/TOTP state machine.
- WorkspaceCoreFeatures.go — UnsafeGenerateUser, CreateWorkspaceAndAssignUser, the transactional core every signup path funnels through.
- PassportActionCreateSessionByMail.go — AuthorizeWithToken, session token minting/reuse.
- BotActions.go — full bot lifecycle.
- TokenActions.go — generic token CRUD (root-only writes).
- CapabilityActions.go, CapabilitiesTreeActionImplementation.go — the permission catalog and its tree projection for the admin UI.
- WorkspaceMembersHelpers.go — the requireRealWorkspaceMembersCapability defense-in-depth check.
- AddUserToWorkspaceActionImplementation.go, RemoveUserFromWorkspaceActionImplementation.go, ChangeUserWorkspaceRoleActionImplementation.go, AcceptInviteActionImplementation.go — the full membership-mutation surface, root-only and self-service.
- PublicJoinKeyActions.go, PublicAuthenticationActions.go — the entity side of the join-link feature and the cross-request scratch-state entity.
- NotificationActions.go, messaging/*.go — notification sending, template compilation,

provider dispatch.

- WorkspaceCli.go, UserActions2.go — where the ws/user CLI command trees are assembled.

10 The Storage Module

modules/storage implements the **tus resumable upload protocol** (Go package historically named fileupload internally — the module name string passed to StorageModuleSetup is still "fileupload", a leftover of an in-progress rename, not two different modules).

10.1 Where the bytes go

Uploaded content is stored as **PostgreSQL large objects** (lo_*), with upload bookkeeping — offset, metadata, completion state, owner, claim state — tracked in a tus_uploads table. The HTTP/storage layer is plain Go and pgx, with no emi-generated entities involved — and, more precisely than "documentation-only," there is *no* StorageModule3.yml anywhere in the module at all. This is a deliberate exception worth stating plainly: unlike wallet or eventbus, storage is a standalone, hand-written Go library wired in through StorageModuleConfig/ModuleSetup, with no entity/action YAML driving it. Its own package doc says as much: it depends only on tusd's handler package and pgx — "nothing here goes through Fireback or Emi."

Storage module map

- Store.go — tus DataStore over large objects
- Handler.go — mounts tus endpoints, enforces auth/quota
- Claim.go — attach/detach an upload to a feature
- Reaper.go — sweeps orphaned, never-claimed uploads
- Quota.go — per-user byte totals & limits
- Download.go — range-request downloads
- Backend.go / MultiBackend.go — pluggable replication targets (disk, S3, MinIO)

10.2 The Store interface and pgLoStore

The extension point is the Store interface: it embeds tusd's handler.DataStore + handler.TerminaterDataStore and adds this module's own bookkeeping surface — OpenRange, OwnerOf, ListFiles/ GetFile/ClaimFile/ReleaseFile, UsedBytes/WorkspaceUsedBytes, orphan-candidate lookup, and backend-selection persistence. Three concrete implementations exist — pgLoStore (Postgres large objects), plus SQLite and MySQL variants — so no other file in the package needs to type-switch on the backend in use.

Each upload is one row in tus_uploads plus one Postgres large object. NewUpload opens a large object and inserts the bookkeeping row inside a single transaction, so the two can never disagree about existence. WriteChunk is the critical per-PATCH-request path: it opens a *fresh transaction for the duration of exactly one chunk*, seeks the large object to the client-declared offset, copies the request body into it, and atomically updates upload_offset/ completed/completed_at in the same transaction. Large objects in pgx can only be manipulated inside the transaction that opened them, which is why every read/write path opens its own short-lived large-object handle rather than holding one open across the request lifecycle; a reader returned for downloads keeps its transaction open until Close() is called — Close *is* the commit.

Reserved Upload-Metadata keys are how the caller's authenticated identity is smuggled from Handler.go's gin layer into NewUpload without trusting the client: they are stripped back out of the metadata before persisting, so they never leak into the client-visible metadata JSON an info response returns.

10.3 Auth and quota, enforced before tusd sees the request

Mount/MountProfile wire the tus protocol's POST/HEAD/PATCH/GET/DELETE onto a gin router. Because tusd's own routed handler expects the mount prefix already stripped from the path, the mounting closure manually rewrites the request path before delegating — plain gin.WrapH doesn't do that rewriting on its own.

Auth/ownership enforcement happens before tusd is ever invoked: a configurable Authenticate callback (nil means every request is anonymous, no ownership recorded, quota never consulted — the module's original behavior, kept as the default) resolves an auth context with user id, workspace id, and access level. For any request against an

existing upload id, the resolved caller is checked against OwnerOf and a mismatch returns 403. For a bare creation POST, the resolved identity is injected into the Upload-Metadata header, overwriting anything the client itself sent under the reserved keys so ownership can't be spoofed.

Quota is enforced as tusd's PreUploadCreateCallback — rejecting the upload *before any bytes are written* if the caller's existing used bytes plus the declared size would exceed their quota. Quota.go defines a 2 GiB default used whenever no explicit quota is configured; a negative quota result means unlimited. Used bytes are summed over the declared size column for every upload row belonging to a user — completed *and* in-progress, since an in-progress upload has already reserved that much large-object space; abandoned reservations are only reclaimed once the reaper runs.

A separate robustness detail lives in the same file: the PATCH request body is wrapped so a client that goes silent mid-chunk (dead wifi, a sleeping laptop) trips a read deadline rather than blocking the copy — and the database transaction holding the large object open — forever. tus-js-client's own retry logic turns the resulting error back into an automatic retry on the client side.

10.4 Claiming: turning an anonymous upload into a real reference

The module's own doc comment names the problem directly: a tus endpoint that needs no prior authorization to create files is "free anonymous file hosting to anyone who finds it" unless something ties an upload's lifetime to a real record. The fix is a two-step contract. ClaimFile takes a row lock, requires the upload be completed (only a finished upload has bytes worth keeping), and sets claimed_by/claimed_at. The current Postgres implementation has actually disabled the "already claimed by someone else" rejection — claiming twice, even under a different claimant, is now a no-op success rather than an error. ReleaseFile clears a claim only if the caller's claimant string matches what's recorded, so one owner can't release another's claim. claimedBy is an application-defined string (e.g. "score:" + uniqueId) — there is no formal foreign key, just a text tag a feature stamps on once it has committed to using the upload.

10.5 The reaper

SweepOrphaned deletes two disjoint sets in one pass: uploads that are *completed but never claimed* past a configurable TTL, and uploads that *never finished* (abandoned mid-transfer) past a separate, typically shorter TTL. Deletion goes through the exact same terminate path a DELETE request uses, so both the bookkeeping row and the backing large object are removed uniformly regardless of vendor. StartReaper runs the sweep once immediately, then on a ticker until its context is canceled — this is opt-in, wired once per process alongside Mount, with no default interval baked into the module itself.

10.6 Downloads: a real range-request subset

MountDownloads registers read-only routes on a separate base path from the tus mount itself, since tusd's own catch-all route can't coexist with a second catch-all at the same prefix. The handler implements a real RFC 7233 subset: single-range Range/If-Range requests are honored against an ETag derived from the upload id and size; a multi-range request deliberately degrades to serving the whole file with 200 (permitted by the RFC, and in practice download managers split a file across several single-range connections rather than issuing one multi-range request). Content-type and disposition mirror tusd's own filtering logic: an unrecognized or missing declared MIME type always falls back to application/octet-stream with an attachment disposition — specifically to stop a maliciously crafted metadata value (e.g. claiming text/html) from becoming a stored-XSS vector when a raw download link is opened directly in a browser. A small, inline-safe allowlist (video/audio/image, minus image/svg+xml, which can itself carry a <script>) gets inline disposition instead.

10.7 Pluggable backends

Backend is a second extension interface — a place upload bytes can physically live on top of, or instead of, the database's own native storage — with disk, S3, and MinIO implementations shipped. MultiBackend.go's wrapper decorates any Store once backend selection is configured: every upload always transits through the database first (so a Backend implementation only ever handles a complete blob at offset zero), then gets replicated to whichever backends are selected, with the database's own transient copy trimmed once repli-

cation succeeds. Reads try each recorded backend in order until one has the bytes. With nothing configured, this wrapper is skipped entirely — zero overhead, matching the module's original single-database behavior.

10.8 Lifecycle, summarized

A client uploads via the tus protocol; once complete, some feature "claims" the file (moving it out of the orphan pool the reaper would otherwise delete). Downloads support HTTP range requests. Quota and ownership are enforced per user/workspace, and everything above is opt-in: only a project whose main.go calls storage.StorageModuleSetup (…) gets these routes and CLI commands mounted.

11 Backup & Restore

modules/backup provides point-in-time Postgres backup/restore built on **wal-g** (github.com/wal-g/wal-g/cmd/pg, a real go.mod dependency, not a shelled-out binary) rather than pg_dump. Where pg_dump produces discrete snapshots that scale with total database size, wal-g decouples restore granularity from database size: a periodic base backup plus every archived WAL segment lets you replay forward to any instant.

> *Fireback's backup module is an orchestration layer, not a reimplementation — it drives wal-g for the actual backup/restore work and adds a local catalog of verify/prune history, automatic nearest-backup resolution for a target point in time, restore-drill automation, and monitoring-friendly health checks.*

11.1 Two caveats worth printing above the fold

1. **Same-disk storage is not disaster recovery.** Local/mounted storage on the same disk as the database only protects against logical mistakes (a bad migration, an accidental delete) — if the host or volume dies, the backup dies with it. Real DR needs WALG_FILE_PREFIX pointed at physically separate storage, or an S3-compatible target.
2. backup push **needs local filesystem access** to Postgres's data directory, not just a network connection — wal-g's remote/streaming mode doesn't support delta backups, so push/verify/prune must run co-located with PGDATA.

11.2 Embedded wal-g via re-exec, not a subprocess binary

wal-g is compiled directly into the fireback binary — there is no separately installed wal-g executable anywhere on the host. The module still shells out via os/exec, for a specific reason: wal-g's own command tree isn't safe to invoke more than once per process, so every operation gets its own fresh process by **re-executing this same binary** with a hidden marker argument, recognized early in the process's own startup and dispatched straight into wal-g's entry point before any of fireback's normal bootstrap runs. Long-running operations like a base backup stream their stdout/stderr straight through so progress stays visible; backup-list –json is captured instead, since wal-g's diagnostic logging goes to stderr and JSON output stays clean on stdout.

11.3 Push, verify, prune, restore

- **Push** — backup-push (with –full to force a full backup; otherwise delta chaining depth is governed by WALG_DELTA_MAX_STEPS). The same re-exec trick backs WAL push/fetch, i.e. Postgres's own archive_command/restore_command entries point at this binary, not a separate wal-g install.
- **Verify** is a full restore drill, and the module's own doc comment calls it "the single highest-value reliability check in this whole module — an unverified backup strategy is not a backup strategy." It wipes a scratch directory, fetches the latest backup, writes a bare restore command (replay everything, auto-promote), starts a real Postgres via pg_ctl on a scratch port deliberately distinct from a live primary's, and waits for recovery.signal to disappear — Postgres itself deletes that file on promotion, which is simpler than authenticating a client connection just to poll pg_is_in_recovery(). A separate, cheap, cron-friendly check command instead compares the newest backup's age against a threshold and inspects wal-g's own WAL-integrity verification, returning a pass/fail plus reasons for alerting.
- **Prune** keeps the newest *N* full backups plus whatever deltas/WAL segments are needed to restore forward from the oldest one retained.
- **Restore** accepts an optional target time, a target directory, a scratch port, and a promote flag.

11.4 Nearest-backup resolution for a point in time

Resolving which backup to restore from for a given target instant is a linear scan over wal-g's own backup list — not a binary search, since the list is the only source of truth and tends to be small — picking the **newest backup whose creation time is at or before the target**. If every available backup is newer than the target, restore refuses outright rather than guessing ("oldest available backup is newer than the requested target"). With no target time given, the newest backup is used for a full catch-up restore. Once a base backup is fetched, restore either writes a specific recovery target (target time plus a recovery action of pause unless promotion was explicitly requested — deliberately leaving the cluster paused in recovery so an operator can inspect data before committing) or a bare replay-everything command.

A subtle, documented bug fix lives in how that recovery target time gets formatted: it uses fixed 6-digit fractional seconds rather than Go's default trimmed-trailing-zeros formatting, because the trimmed form could silently truncate sub-second precision and exclude a transaction that happened earlier in the same second as the real target. The restore command written into postgresql.auto.conf again points at the *same running binary* via the re-exec marker — Postgres can only fork/exec a real path on disk, so there is still no second wal-g binary anywhere, even mid-recovery. Starting the recovering instance also disables archive_mode, since a recovering copy must never archive WAL back into the same prefix its own base backups came from.

11.5 Offsite copies

A dedicated copy operation produces a self-contained, restorable copy of one named backup at a different storage prefix, without decrypting or recompressing — using two throwaway JSON storage-config files under the hood, since wal-g's copy command doesn't accept its configuration via environment variables the way other subcommands do. An optional "with history" mode additionally copies every WAL segment from that backup's own recovery point forward through the current moment, making the copy restorable to any instant up to when the copy ran, not just to the backup's own completion instant.

11.6 The plain dump/restore-dump path

This is a deliberately independent, simpler mechanism — "just get me a copy of this one database" — covering all three vendors fireback supports, not only Postgres. Postgres uses pg_dump -Fp –no-owner –no-privileges (no extra single-transaction flag needed, since pg_dump already always runs inside one REPEATABLE READ transaction internally). MySQL uses mysqldump –single-transaction –routines –triggers –set-gtid-purged=OFF — the last flag was added after discovering that restoring a GTID-enabled dump into a database with its own GTID history fails with an overlap error. SQLite uses VACUUM INTO, atomic and safe against concurrent readers/writers by SQLite's own backup-API guarantee. Every vendor's output is wrapped in a single-entry zip streamed directly to whatever writer the caller supplied — a file, or an HTTP response — with no double-buffering through disk. Restore-from-dump refuses to overwrite an existing target unless forced; the SQLite path additionally takes a timestamped backup copy of the existing file first and runs PRAGMA quick_check against the restored file before the final atomic rename, refusing to commit if the check fails.

Large objects need no special-case code

> modules/storage's uploaded files (§10) live as Postgres large objects, and modules/backup has *no* large-object-specific Go code anywhere — pg_dump includes large objects by default unless –no-blobs is passed, which this module never does. tus_uploads' large-object reference still resolves correctly against whatever OID Postgres assigns on restore. The whole integration is really just "don't pass –no-blobs," not a bespoke code path — a good example of two modules composing for free because neither one special-cases the other.

11.7 HTTP hash-streaming endpoint

A dump can also be requested over HTTP: one route registers a one-time job, a second streams it. Neither endpoint carries its own bearer-token auth — an explicit config-based API token requirement existed once and has since been removed. Access control today is purely the download **hash itself as the sole credential** on the GET side, and OS-level file

privacy on the registration side: job files are written under a per-OS user config directory with restrictive permissions, so registering a job in the first place already requires being the same OS user the server process runs as — "whoever can write there is already as trusted as whoever runs backup dump locally." Each job is a small JSON file named after a securely random hash, read and immediately deleted the moment it's claimed — single-use, disabled the instant it's fetched, successfully or not — with missing, corrupt, and expired jobs all treated identically as "not found" so a caller can't distinguish a stale hash from a wrong one. Expired jobs are swept opportunistically whenever a new one is registered, rather than needing a dedicated background goroutine. The dump itself only executes when the GET actually fires, streaming straight into the HTTP response — the server never writes the dump to disk.

11.8 Configuration: retention, not scheduling

The module's emi config block declares plain fields folded into fireback's combined config list: the wal-g file prefix (required), compression method (default lz4), delta-chain depth (default 6), how many full backups to retain (default 4), the dump directory, a dump-hash TTL (default 1800 seconds), and the four dump/restore client binary paths. **There is deliberately no scheduling config at all** — no cron expression, no interval field; the check command's own usage string ("fast health check for cron/monitoring") confirms the intended operating model: an *external* cron job invokes backup push/prune/check on a schedule the operator owns, and the module itself never starts a background scheduler — mirroring wal-g's own co-located, cron-driven operating model.

12 The Event Bus

modules/eventbus gives Fireback modules a publish/subscribe channel, defined declaratively (EventBus.emi.yml) like other emi entities. Two backends ship out of the box, both implementing the same small interface (AddUser/RemoveUser/ ListUsers/ IsUserIn/FireEvent/Subscribe):

- **Local** (EventBusLocal.go) — in-process delivery, useful for single-instance deployments and tests
- **Redis** (EventBusRedis.go) — cross-instance delivery for horizontally scaled deployments

12.1 Local vs. Redis mechanics

LocalEventManager tracks per-instance user membership in a *package-level* map guarded by a package-level mutex — deliberately package-level rather than per-instance, because a websocket connect and a different connection's disconnect cleanup run on different goroutines and genuinely race in normal operation; an unsynchronized concurrent map write in Go is a fatal crash, not a benign data race, so this lock is not an optional nicety. FireEvent doesn't touch that membership map at all — it publishes onto an in-process event bus library under a fixed topic, and Subscribe registers a listener on that same topic that JSON-decodes the payload and routes it onward. Because that underlying library registers listeners synchronously, StartEventBus subscribes the local backend *in the foreground*, guaranteeing an event fired right after startup returns is already routable — no "a moment before subscriptions are live" window to worry about.

RedisManager is a thin wrapper over real Redis primitives: user membership maps onto a Redis set per instance (SADD/SREM/SMEMBERS/ SISMEMBER), FireEvent publishes onto a Redis pub/sub channel, and Subscribe blocks forever in a receive loop over that channel — since this blocks, StartEventBus runs it in a goroutine instead, and explicitly documents the resulting small async window before the subscription is genuinely live.

StartEventBus(redisEventsUrl) is the switchboard: an empty URL keeps the local backend; a non-empty one tries Redis and silently falls back to local on connection failure. It runs from the module's OnAppStart hook rather than unconditionally, so "a project only pays for it if its main.go actually calls eventbus.ModuleSetup(...)" — consistent with the opt-in module philosophy in §3.1.

12.2 The websocket layer

Socket.go is minimal by design: a shared connection pool keyed workspace → user → list of connections, guarded by its own package-level mutex. Each stored connection carries a *snapshot* of the caller's per-workspace access, taken at connect time — this is what later lets event fan-out do a per-connection permission check without re-querying the database for every single event.

12.3 Fan-out: routing an event to the right sockets

Once a message round-trips through whichever backend's pub/sub, a routing function walks every open connection, skips any workspace that doesn't match the event's own source workspace, and for each remaining connection in the right workspace:

- if the event catalog entry declares required permissions, the connection's stored access snapshot is flattened into user/workspace capability lists and checked with the same MeetsAccessLevel machinery §9.2 describes, delivering as soon as any permission group passes;
- delivery itself is a plain JSON websocket write, with write failures logged rather than torn down synchronously.

A documented, previously-shipped bug is worth keeping in mind here: an earlier version left the connection's user/workspace capability lists both nil rather than deriving them from the stored access snapshot, which meant permission-gated events were silently denied to *every* connection, unconditionally — fixed by resolving them explicitly per connection, per event, rather than assuming they'd already been populated elsewhere.

12.4 method: reactive: what gets generated differently

Subscription lifecycle is itself a first-class emi action — EventBus.emi.yml declares exactly one, with method: reactive instead of get/post. Compared to a normal action, the generated Gin handler is a **websocket-upgrade handler**, not a function returning JSON:

```
actions:
  - name: EventBusSubscription
    description: >
      Connects a client to all events related to
      their user profile, or workspace they are in
    method: reactive
    url: /ws
```

The generated code builds a session wrapping the upgraded connection, spawns one goroutine pumping inbound frames into a read channel and another draining a caller-returned write channel back out onto the socket, closing cleanly whichever side finishes first. The developer-supplied implementation is handed that session and returns a write channel plus an error — the fundamental generated-code difference from a normal action, whose handler signature is a single request in, response out. A reactive action's handler instead owns a live duplex channel pair for the connection's entire lifetime. The compiler also generates a symmetric *client*-dial helper — something a plain HTTP action never gets — that rewrites http(s):// to ws(s)://, carries the same TLS/mTLS options as any other generated client call, and exposes the identical read/write/-done channel shape on the client side, so a Go client of the same SDK can dial a reactive endpoint symmetrically. Internally, the action's declared method is literally the string "REACTIVE" rather than an HTTP verb — that string is what routes the emi compiler into this whole alternate code-generation path in the first place. The internal-stats stream in §14 reuses the identical convention.

The actual subscription logic resolves the caller's auth context the same way any other action would, registers the connection into the socket pool and the chosen event-bus backend, and runs its own read loop distinguishing normal websocket close codes from real errors — with a documented fix for a break that used to only exit an inner select rather than the surrounding loop, which let a single disconnect run the socket-pool cleanup twice concurrently for the same connection — exactly the class of bug the local backend's membership map lock (§12.1) also exists to prevent.

13 Finance & Wallet

modules/finance groups Fireback's money-handling features:

- **wallet** — per-user/workspace balances and ledger entries, the newest major area of active development in this repository
- **walletpublic** — the subset of wallet operations exposed to end users versus internal/administrative use
- **fakepayment** — a stand-in payment provider for local development and tests, so wallet flows can be exercised end-to-end without a real payment gateway configured

As with every other module, wallet functionality only appears in a build once wired into main.go; the fake payment provider keeps that wiring cheap during development, with a real gateway swapped in for production the same way abac's authorizer is swapped in for other modules.

13.1 Balance model: a running total, not a derived sum

A wallet's balance is stored as a decimal-minor-units string, mutated only inside a locked database transaction and mirrored by a ledger row on every change — so the balance is a *cached, running* total, not something recomputed by summing ledger rows on every read. The ledger exists purely for audit history and idempotency; the wallet row itself is the fast-path source of truth for "what can be spent right now." A version counter increments on every mutation as defense-in-depth, but the real concurrency guard, described next, is a database row lock — the version counter is explicitly documented as secondary to that.

13.2 applyLedgerEntry: the one mutation path

The module's own comment states the invariant directly: this is the *one* place a wallet's balance is ever mutated; every path that moves money goes through it. Its algorithm:

1. An idempotency short-circuit: if a ledger row with this request's idempotency key already exists, return it unchanged — checked *before* opening the mutating transaction, so a safe retry never even takes a lock.
2. Inside one transaction, take a row lock on the wallet (SELECT … FOR UPDATE) — this is the actual concurrency guard, serializing concurrent mutations against the *same* wallet.
3. Reject if the wallet isn't active, or is frozen (a root-only flag the owner cannot clear themselves, distinct from the owner-settable status field).
4. Compute the new balance using arbitrary-precision integer arithmetic — deliberately never float64, since floats are lossy for amounts with many decimal places and "money must never round in ways nobody asked for." A debit that would go negative returns a plain not-ok result, not an error — an expected business outcome, not a malformed request.
5. Update the balance/version and insert the ledger row in the *same* transaction, so ledger and balance can never drift apart.
6. Belt-and-suspenders: if two callers race past step 1 with the same idempotency key, the loser's insert trips a unique index and is caught and turned back into "return the existing row," without inspecting driver-specific error codes that differ across Postgres/MySQL/SQLite.

Every money-moving action is a thin wrapper around this one function: purchases (debit), a root-only manual balance adjustment (requiring a note for audit), treasury funding (root-only credit), a payment gateway's webhook handling (credit, reusing the payment attempt's own idempotency key so a duplicate webhook delivery can never double-credit), and prepaid redemption.

A transfer between two wallets composes *two* applyLedgerEntry calls (debit source, credit destination) inside one outer transaction, and sidesteps a real deadlock hazard: since each call locks one wallet, two concurrent transfers moving funds in opposite directions between the same pair of wallets would deadlock if they acquired locks in different orders. Transfers always issue their two calls in a fixed order — whichever wallet's identifier sorts first lexically — regardless of which leg is the debit and which is the credit.

13.3 Treasury: funded pointer wallets

A treasury entity is, in the code's own framing, purely a named pointer to an existing wallet — creating or updating a treasury row never moves money; only an explicit funding action does, crediting the treasury's linked wallet through the normal ledger path. A treasury can then back a prepaid gift card: redeeming a treasury-backed card debits the treasury's wallet *before* crediting the destination wallet, which is what makes redemption a real transfer instead of minting value out of nowhere — if the treasury can't cover it, the same insufficient-balance check that guards ordinary debits fails the whole redemption before the destination wallet is ever touched. An *unbacked* card (no treasury attached) skips this and mints the redeemed value from nothing, matching every prepaid card's behavior before the treasury concept existed — kept deliberately as a simpler fallback rather than being retired.

13.4 wallet vs. walletpublic

wallet is the root/admin-facing module: full CRUD over wallets, treasuries, provider configs, gateway config, and actions like balance adjustment or freezing a wallet, gated by explicit permission keys or

root-only access. walletpublic is the self-service, owner-facing surface, and owns *no entities of its own* — every action is gated purely by "logged in, and you own this wallet," not by a permission key at all. Ownership is resolved across three owner shapes (a user, a workspace, or a specific user-within-a-workspace), reusing abac's existing membership table rather than reimplementing membership checks. walletpublic's response DTOs are separate, hand-maintained projections that intentionally omit internal-only fields — the split exists specifically so a future internal-only field added to wallet doesn't leak into the public API by accident, rather than by the projections happening to be safe today.

13.5 fakepayment: a real gateway's shape, no network call

fakepayment implements the exact same gateway-adapter interface a real provider does (initiate payment, verify webhook, check status) — confirmed by real Przelewy24 and ZarinPal adapters living alongside it under the same interface. Its "initiate payment" never talks to any external network; it builds a redirect URL to its own embedded checkout page instead:

```
checkoutUrl := cfg.BaseUrl +
  "/fakepayment/checkout?ref=" + attempt.UniqueId
return &GatewayInitResult{
  GatewayReference: attempt.UniqueId,
  RedirectUrl: checkoutUrl,
}, nil
```

That checkout page is served from an embedded static HTML file — plain HTML and vanilla JS, deliberately *not* part of the React admin frontend, matching how a real gateway's own hosted checkout page is an entirely separate surface from the app itself. The fake card/BLIK "pay" actions complete the payment attempt *synchronously in-process* by calling the exact same webhook-processing function a real webhook handler would call, rather than faking an actual HTTP round trip back through the gateway webhook route. Its status-check always returns nothing, since there's no external system to reconcile against.

This is the cleanest illustration in the codebase of how a real gateway gets swapped in: a gateway registry is a map from a short code to an adapter satisfying the four-method interface, and a new provider is just a new struct registered under its own code — new providers plug in without ever touching the emi schema. The webhook route itself is a deliberately hand-wired plain Gin route, not emi-generated, because a webhook arrives with no caller session and needs the raw request body and headers for signature verification; it normalizes both POST-body gateways and GET-redirect gateways with no server-to-server callback at all (ZarinPal, whose query string gets flattened into a JSON object) into the same shape before handing off to the adapter's own verification.

All minor-units arithmetic across the module funnels through one small file backed by arbitrary-precision integers — the only place that touches the string representation of an amount directly, keeping every other file working with parsed values instead of ad hoc string math.

14 Internal Stats

modules/internalstats exposes read-only server/process health — CPU, memory, disk, network, and this process's own Go runtime — around forty individual measurements collected by gopsutil (github.com/shirou/gopsutil/v3), cross-platform on linux/darwin/windows.

> ***Why this exists:*** *every module in this book leans on Fireback running as one static binary, so there's no separate metrics agent (node_exporter, a sidecar, …) to bolt on if an operator just wants to know whether this process itself is healthy. Internal Stats is that answer built in — the same binary that serves requests also reports its own CPU/memory/disk/network/goroutine numbers, over HTTP, websocket, and CLI, gated behind the root workspace so it's safe to leave reachable without a separate monitoring stack.*

14.1 The dashboard

/manage/internal-stats (InternalStatsDashboard.tsx) is the plain-HTTP-snapshot renderer above, polled every two seconds — the screenshot below is captured live by e2e/cypress/e2e/manage-internalstats.cy.ts the same way §6's login figures are, so it always shows the real current UI rather than a hand-captured image that can drift out of date. The spec captures both palettes in one run (toggling body's dark-theme class directly, the same override the UI's own theme switcher sets), and this section picks whichever one matches — light or dark — to match the palette of the PDF currently being built (\iffirebackdark, see main-

body.tex).

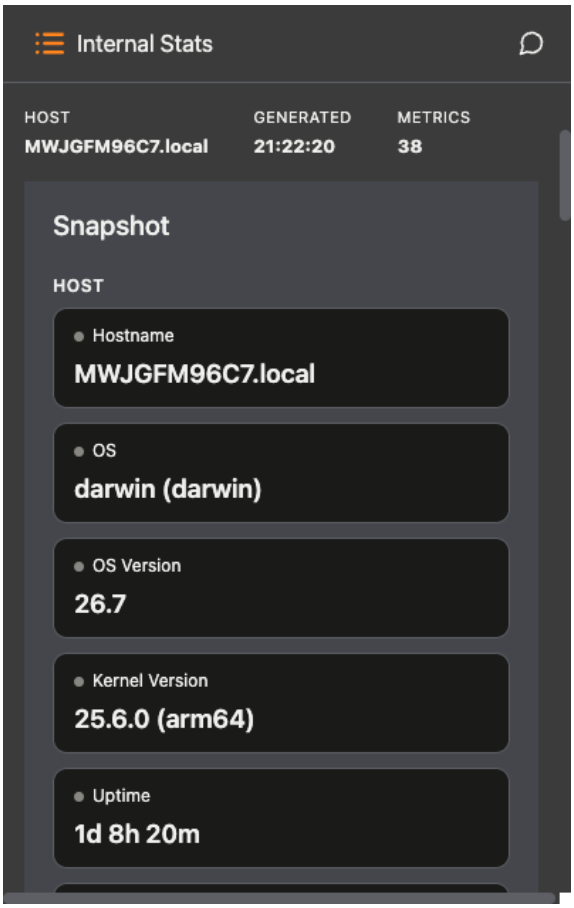

Figure 4: The Internal Stats dashboard, root-only, live-polled every 2s.

One snapshot, four renderers

- GET /internal-stats/snapshot — one-shot JSON
- GET /internal-stats/stream (websocket, method: reactive) — the same shape, pushed on an interval
- internalstats snapshot (CLI) — one-shot JSON, same auth path as the HTTP route
- internalstats watch (CLI) — a live, colored two-column terminal table, consuming the stream in-process with no network hop or Authorize check — the same trust boundary as backup dump / config list: an operator already holding a shell on the box

Collector.go's CollectSnapshot is the single source of truth every transport above renders — there is exactly one place that knows how to read a stat, and four thin wrappers around it.

14.2 What gets measured

The package doc states the scope directly: around forty measurements across host, CPU, memory, disk, network, and this process's own Go runtime. Counting the actual fields confirms the shape: host information (hostname, OS, platform version, kernel version, uptime, boot time, process count — seven), CPU (model, MHz, physical/logical core counts, used percent, 1/5/15-minute load averages — eight), memory (total, used, free, available, used percent, cached, plus swap total/used/used percent — nine), disk (total, used, free, used percent — four), network (bytes and packets sent/received — four), and Go runtime (version, goroutine count, heap allocated, heap from system, GC count, process uptime — six). All of the OS-level numbers come from gopsutil rather than a hand-rolled per-platform split — explicitly contrasted, in the module's own comments, with the backup module's manual unix/windows split for disk-space checks, since gopsutil is already cross-platform and there's no reason to repeat that split across five more metric families.

Two vendor quirks are worked around directly: Apple Silicon's CPU info call returns a bogus, tiny clock-speed value from sysctl, so it is only reported when at least 100 MHz (filtering out the bogus reading without hiding a genuinely low value on other hardware); and load averages are unix-only in gopsutil, reported as unavailable rather than failing the whole snapshot on Windows.

CollectSnapshot does no caching at all — every call is a genuine point-in-time read, including a live, deliberately short 200ms CPU sample so it doesn't stall any caller noticeably. Severity is computed generically for any percent-based stat from two fixed thresholds (warn at 75%, critical at 90%); everything else is plain informational, since — as the code comments note — there's no meaningful "too high" reading for, say, a hostname or a Go version string.

14.3 One collector, four renderers

Stream.go's streaming primitive pushes the first snapshot immediately (not after waiting a full interval), then on a ticker, closing its output channel once the caller's context is done. The module's own comment is explicit that both real consumers of this — the reactive websocket action and the CLI watch command — "are really just two different renderers over this same feed rather than separate implementations of 'collect on an interval'." Concretely:

1. **HTTP snapshot** — a plain request/response emi action: authorize once, return one CollectSnapshot() call wrapped in the standard single-item response envelope.
2. **HTTP stream** — the same method: reactive convention §12.4 covers for the event bus: authorize once per connection, then relay the streaming primitive's output as JSON frames over the generated reactive channel plumbing, canceling its own streaming context the instant the session's done signal fires rather than leaking a ticker goroutine until the next tick discovers the write side is already gone.
3. **CLI snapshot** — generated for free straight from the emi action definitions; no bespoke CLI code is needed for a one-shot read.
4. **CLI** watch — hand-written, layered on top of the generated CLI: subscribes to the streaming primitive directly in-process (no network hop, no websocket, since the CLI already runs on the box being measured) and redraws a colored, category-grouped label/value table on every tick, colored by the same severity the collector assigns.

> *The HTTP snapshot and stream actions both require the caller's token to belong to the root workspace specifically — deliberately stricter than other reactive modules' workspace-scoped defaults, since server health data is sensitive (hostnames, indirectly IPs, resource pressure an attacker could use to time an attack). The CLI* watch *command, by contrast, calls no authorization function at all: like every other local admin command in this codebase (*backup dump, config list, *...) it doesn't go through that gate, because the gate exists for the HTTP/websocket routes, not for an operator already holding a shell on the machine. The same sensitive data that requires root-workspace auth over the network is unconditionally available to anyone who can execute the CLI binary on the host — a deliberate design choice, not an oversight, but one worth knowing when reasoning about container or host isolation boundaries around this binary.*

15 AI Chat & Tool Use

modules/ai adds a chat feature backed by a swappable large language model backend — Claude or ChatGPT out of the box — with the model able to call back into the application's own actions as *tools*. The interesting part isn't the chat widget: it's that every tool the model can call is generated, not hand-written, straight from the same declarative *.emi.yml definitions every other module in this book already uses for its HTTP routes and CLI commands.

modules/ai **at a glance**

- GET /ai/prompt — the chat endpoint, method: reactive (websocket), same convention §12.4 covers
- /ai/mcp — a real MCP (Model Context Protocol) server, mounted on the same HTTP server, streamable-HTTP transport
- ai mcp serve — the same MCP server again, over stdio, for a local MCP client
- aiProviderConfig — one entity, root-only, holding each provider's settings as a JSON blob (API key, model, temperature, ...)
- ai.usage / ai.manage — the two capabilities gating chat use and provider configuration respectively

15.1 A swappable backend, configured only in the database

LLMProvider is a single-method interface (StreamChat(ctx, messages, tools) (chan StreamChunk, error)) that Claude's and ChatGPT's own request/response shapes are each adapted to — the same role wallet.GatewayAdapter (§13) plays for payment providers. modules/ai itself never imports either vendor's concrete package: a separate modules/ai/providers package is the one place that does, wiring both into a map[string]ProviderFactory keyed by providerType ("claude"/"chatgpt").

Which provider is actually active is resolved fresh on *every new* chat connection — not once at process start — by querying the aiProviderConfig table for the first non-disabled row whose providerType has a registered factory. An administrator flips providers, or retunes one's temperature/model/max-tokens, purely from the AI settings screen; nothing about that change needs a redeploy or a main.go edit.

There is deliberately no environment-variable fallback for provider credentials. A provider factory starts from a zero-valued config struct and only ever fills it from that row's own JSON column; if no API key is set there, the factory returns a plain error and the connection fails cleanly at connect time — never a live request reaching Anthropic or OpenAI with an empty credential and bouncing back a confusing 401.

15.2 From an emi intent to a model-callable tool

A module exposes one of its existing actions to the model by declaring an intents: entry alongside its usual actions: block — no new endpoint, no new business logic, just a tool signature derived from something that already exists:

```
intents:
  - name: editUser
    from: userUpdate
    title: Edit user
    annotations:
      destructiveHint: true
      idempotentHint: true
```

The emi compiler's GoIntentsGenerate resolves from: against the module's own actions and, for each intent, emits an <Name>IntentMeta() accessor carrying a name, description, and a single, flat JSON Schema — InputSchema — built by merging *everything* a tool call would otherwise need to know separately: the action's body (dto or inline fields), its query parameters, its URL path parameters (a bare :uniqueId in the route becomes a required schema property), and any typed request headers, each converted through the same field-to-JSON-Schema builder. A model calling editUser sees one argument object; it never needs to know that uniqueId actually travels as a URL path segment on the real HTTP route while everything else is body.

Not just metadata

For every intent derived from a real action, the compiler also emits a ready-to-call Register<Name>IntentTool(reg, handler) function — the tool-calling equivalent of a generated <Action>Gin/ <Action>Cli wrapper. It builds the tool's schema, unmarshals a call's JSON arguments into the exact dto/path-parameter/query shape the real handler expects, calls it, and marshals the response back — reg is the shared registry described next, and handler is the same Go function (abac.UserUpdateAction, unchanged) the HTTP route already calls.

An intent with no from: — declared standalone, with its own inline in:/out: — still gets a full IntentMeta and schema, but no Register...IntentTool: there is no resolved Go function to wire it to automatically, so a project wires that one by hand if it needs it.

15.3 Tools belong to the module that defines them

A project never hand-assembles the model's tool list. Any application.ModuleProvider can set AiToolsProvider, a function that adds its own tools onto a shared *emigo.ToolRegistry — the same append-your-own-piece pattern already used for a module's permissions, config, or CLI commands. The ai module walks every registered module's AiToolsProvider once, at server start, and combines the results; a module's tools become reachable the moment that module is registered in main.go, with nothing tool-specific left to write there.

Hand-listing every Register...IntentTool call inside a module is still one line per tool, though, and emi closes that gap too: a manifests: block with types: [go-mcp] bundles *every* from:-derived intent in the module into one generated <Name>McpToolManifest(reg) function — the MCP equivalent of a go-cli manifest's <Name>CliManifest. abac's own module setup ends up as one line:

```
AiToolsProvider: abacdefs.AbacMcpToolManifest,
```

Declaring a new tool is then genuinely just: add the intents: entry, recompile. No file in cmd/fireback changes at all.

15.4 Two transports, one tool list

The exact same *emigo.ToolRegistry backs two independent ways of reaching a module's tools:

1. **In-process chat** — the /ai/prompt websocket streams the model's reply token by token, and whenever the model asks for a tool, the connection's own already-authenticated request context is reused to call it — the same resolve.ResolveActionContext call UserUpdateAction's real HTTP route would make, run again

for the tool call, with no separate auth path of its own.

2. **A real MCP server** — the same tool list, exposed as actual MCP tools, over both a streamable-HTTP endpoint (/ai/mcp, mounted on the app's own HTTP server) and stdio (ai mcp serve, for a local MCP client). Unlike the chat websocket, an MCP call has no ambient session to inherit authorization from, so each tool's schema gains three extra properties at registration time — token, workspaceId, roleId — and every call builds a fresh, call-scoped request context from them before dispatching, exactly mirroring what a real HTTP request's headers would carry.

A generic fireback_cli tool rides alongside the typed ones on both MCP transports: it re-runs the same CLI command tree GetCliCommands already builds, so every action reachable from the command line is reachable this way too, with zero maintenance as new modules and commands accumulate — a deliberate fallback for whatever hasn't been given a typed intents: entry yet.

> *Declaring a tool is cheap, but reachability isn't the same as authorization: every tool call still runs the target action's own* SecurityModel *check, unchanged.* userBrowse*, for example, was tightened to* AllowOnRoot: true *specifically once* listUsers *made it reachable from a chat connection scoped to any workspace, not only while actually connected to the root one — the same rule the equivalent HTTP route now enforces too. Exposing an action as a tool never loosens what that action already required.*

16 Writing Features in emi

Fireback's own generator, **emi** (github.com/torabian/emi, a separate project), turns declarative YAML module definitions into generated Go, TypeScript and OpenAPI. New features in this codebase are expected to be defined in emi *first* — entities, actions, DTOs, permissions — before any hand-written logic is added to fill gaps emi does not cover.

16.1 One type system, one validation convention

Entity fields, entity-scoped actions, and top-level module actions (FirebackEmiAction / EmiActionBody / EmiField) all share the same shape today. None of them carry a validate property directly on the field; required/validation always goes through tags: { validate: required } instead — one convention, applied uniformly, rather than a per-context special case to remember.

16.2 Relations

Entity relations use type: one or type: collection plus target: <Entity>Entity (PascalCase, "Entity" suffix); add module: <moduleName> only when the target entity lives in a different Fireback module than the one defining the relation.

```
fields:
  owner:
    type: one
    target: UserEntity
    module: abac # only if elsewhere
```

> *Before inventing a new DTO, action, or permission key: search* modules/*/*Module3.yml *and the vendored examples under* modules/{payment,suggestion,abac} *in the vendored emi module for an existing shape that already does what's needed, and reuse or extend it. Freehand-authored shapes that duplicate an existing one produce inconsistent DTOs and bypass the generator.*

16.3 Per-project schema copies

Each Fireback-based project keeps its own copy of the Module3 JSON schema, under a .jsonschemas/ directory at the project root — read that copy before editing any *Module3.yml, since it can differ per project/vendor version from the schema shipped inside the vendored fireback Go module. *.dyno.go files sitting next to a *Module3.yml are auto-generated by an external watcher — only ever edit the YAML, never the generated Go.

16.4 Non-entity modules and migrations

Plain (non-entity) modules — like backup and storage above — use **goose** migrations instead of emi-generated schema: – +goose Up / – +goose Down SQL files under a migrations/ directory, embedded via a migrations/index.go with //go:embed *.sql.

16.5 Regenerating the frontend SDK

The TypeScript side of this contract (§17.8) is produced the same way: ./app emi js:sdk ... –tags typescript emits, per backend action, a typed request/response DTO pair, a static class mirroring the action's JSON definition, and a React hook built on the shared transport layer. The root Makefile's sdk-fetch target is what actually runs this compiler

invocation and refreshes ui/packages/js-remote-ctx — the framework-agnostic transport primitives (fetchx, WebSocketX, buildUrl) that every generated hook is built on — itself entirely regenerated output, never hand-edited, exactly like every *.dyno.go file on the Go side.

Two further Makefile targets, interface-manage and interface-ss, close the loop from a fresh UI build to what the server binary actually embeds (§17.12): each builds the relevant Vite app, wipes and replaces the corresponding modules/interfaces/* directory with the new ui/dist output, then runs git checkout on just that directory's index.go so the //go:embed directive itself is never accidentally clobbered by the copy. Regenerating the SDK and rebuilding the embedded UI are two separate steps precisely because they sit on either side of the same emi-generated contract — one produces the TypeScript client code a component imports, the other produces the static bundle a running server actually serves — and neither one implies the other has also been re-run.

17 The React UI

ui/ is the bundled React frontend shipped alongside the server binary. This section covers its architecture, how it consumes the emi-generated backend contract, and how it gets embedded into the Go binary you actually run.

17.1 High-level architecture

The UI is a **Vite 6 + React 19 + TypeScript** single-page application — not Create React App, not Next.js. Routing is react-router-dom v6 (BrowserRouter or HashRouter, chosen per build variant). State/data management rests almost entirely on @tanstack/react-query: every generated SDK action exposes a useXQuery/useXAction hook wrapping useQuery/useMutation against one shared QueryClient. There is no Redux, Zustand, or MobX anywhere in the dependency tree — local/global UI state (locale, sidebar, menu registry, the authenticated session) is plain React Context. Forms run on **Formik**, paired with a custom form-field component library.

Nothing talks to the backend by hand. The emi compiler (the same tool described in §16) is invoked as ./app emi js:sdk ... –tags typescript to emit, per backend action: a typed request/response DTO pair (private fields, typed getters/setters, toJSON, applyFromObject, clone), a static class carrying the action's URL/method/JSON-schema definition, and a React hook built on the shared transport layer. Real-time actions (method: "reactive", §12) generate a WebSocket-based hook instead. The framework-agnostic transport primitives underneath (fetchx, WebSocketX, buildUrl) themselves live in a dedicated package that is entirely regenerated by the same generator and never hand-edited.

17.2 Directory structure

ui/ is an **npm workspace**: packages/* holds roughly sixteen independently versioned @fireback/* packages, mirroring the backend's module-per-feature layout, while src/ holds the (small) per-app entry points and routing that assemble those packages into a deployable product.

ui/src

- apps/ — the three deployable sub-apps: projectname (the full product shell), self-service (standalone auth/account app), wasm-demo (in-browser demo).
- demo/ — a component/module documentation gallery (Demo*.tsx screens paired with *.snippets.ts usage examples) — effectively a documentation site embedded in the same repo, not core product functionality.
- styles/ — global CSS overrides.
- index.tsx — the single Vite entry point (see §17.3).

ui/packages (selected)

- ui-core — shared components (forms, layout, generic CRUD scaffolding, reactive search) and hooks used by every app.
- js-remote-ctx — the generated, framework-agnostic transport layer (fetchx, WebSocketX).
- auth-client — standalone session/workspace context, deliberately dependency-free beyond React.
- enterprise-shell — the app shell wiring everything together (EssentialApp.tsx, WithFireback.tsx).
- selfservice — every auth/account screen.
- manage — admin/back-office screens.
- messaging — mail/SMS provider config.
- wallet / walletpublic — finance module UI.
- resumable-uploader — tus-based upload widgets.
- wasm-server — pglite/wasm bootstrap, used only by the wasm-demo app.

17.3 The "apps" concept

The UI is genuinely multi-app, using **conditional compilation** rather than separate Vite projects. ui/src/index.tsx is the single entry for every build:

```
// #v-ifdef VITE_TARGET_APP == 'projectname'
import App from "./apps/projectname/App";
// #v-endif
// #v-ifdef VITE_TARGET_APP == 'self-service'
import App from "./apps/self-service/App";
// #v-endif
// #v-ifdef VITE_TARGET_APP == 'wasm-demo'
import App from "./apps/wasm-demo/App";
// #v-endif
```

vite.config.ts derives a build's variables JSON from the –mode string, injects VITE_TARGET_APP as an env var, and a conditional-compiler Vite plugin strips the non-matching branches so only one App.tsx's import graph is ever bundled. package.json scripts enumerate the concrete targets: start/build (projectname, in local/bundle/-capacitor/ manage variants), self-service/self-service:build, wasm-demo/wasm-demo:build/ wasm-demo:github.

wasm-demo is architecturally distinct: it wraps the shared EssentialApp shell in a WithWasmServer provider that downloads fireback.wasm + wasm_exec.js and boots an entire fireback server compiled to WebAssembly *in the browser tab*, backed by **pglite** (Postgres compiled to wasm) — no network backend at all (as introduced in §3.14). Once booted, every generated SDK call is transparently redirected to a wasm request handler, picked up in exactly one place in the whole codebase so pglite's multi-megabyte assets never leak into the projectname/ self-service bundles, each of which is its own separate Rollup module graph. The Makefile chain is make wasm (compiles cmd/fireback-wasm to wasm) → make wasm-demo (npm run wasm-demo:build, output in ui/dist) → make wasm-tg (publishes to GitHub Pages).

17.4 Auth and workspace screens

All auth-flow UI lives under ui/packages/selfservice, as paired *.presenter.tsx (logic/hooks) and *.screen.tsx (render) files: a welcome/method chooser, email/phone passport check, OTP verification, TOTP setup/entry, account-creation completion, password sign-in/reset/change, sign-out, a list of a user's linked passports, the public-join-key join flow, workspace-invite issuance/acceptance, member list/remove/change-role, and role editing (RolePermissionTree.tsx, consuming the same capability tree as the admin UI, §17.5).

The routing file splits into a public route table (no session required: welcome, email/phone, TOTP setup/enter, complete, password, OTP, reset password, sign-out) and an authenticated route table (session required: passports, change-password, plus every invite/member/role sub-route). The self-service app decides at runtime, based on whether a local session exists, which table to mount — with reset-password/sign-out deliberately present in *both*, since a shared or emailed link can legitimately need either. Cross-app session handoff runs through @fireback/auth-client's AuthenticationProvider, a standalone, backend-agnostic session/workspace context backed by localStorage: a consuming app's authenticate() simply redirects to the self-service app's login URL and expects a session back via cookie or a page that reads a token and calls setSession() directly.

17.5 Admin and management screens

Under ui/packages/manage, each admin resource follows the same generated-CRUD scaffolding

pattern (list, columns, edit form, entity manager, archive screen, routes): workspaces, workspace types, workspace config (SMTP/OTP settings), users (including a per-user linked-passport view), capabilities, passport methods, notifications (admin-triggered send), messaging-config (SMTP/SMS provider selection), internal stats (a dashboard polling every two seconds rather than using the websocket stream the backend also exposes, §14), and analytics.

Role/capability editing is shared rather than duplicated: RolePermissionTree.tsx is the actual consumer of the backend's CapabilitiesTree action (§9.2) — it fetches the nested capability tree and renders a checkbox tree with tri-state (checked/unchecked/indeterminate) nodes whose change callbacks mutate a flat array of granted capability keys, used by both the admin role editor and self-service's own role screens.

17.6 Wallet and finance UI

ui/packages/wallet is a full CRUD-plus-workflow package: balances, ledger/transaction entries, an audit-trail event view, a manual balance-adjustment screen, payment-provider configuration, a payment attempt / fake-payment flow pairing with the backend's fakepayment module (§13), treasury accounts, prepaid balances, and admin wallet creation. Like manage and selfservice, it carries its own copy of the generated SDK/runtime rather than sharing js-remote-ctx directly. walletpublic mirrors this for the public-facing (unauthenticated payer) surface of the same module.

17.7 Storage and file-upload UI

Resumable uploads are handled entirely client-side by ui/packages/resumable-uploader, a standalone package built directly on tus-js-client. For each queued file it constructs an Upload with endpoint, chunk size, retry delays, a custom HTTP stack, and lifecycle callbacks; auth headers are applied per-request in onBeforeRequest rather than as a static option (a static option plus onBeforeRequest together double-applied the header and produced an unparseable, comma-joined value — a documented fix). It listens for the browser's online/offline events, pausing in-flight uploads and auto-resuming them, and on a hard failure after tus's own retry delays are exhausted it halves the chunk size and retries a bounded number of times before surfacing an error.

A custom HttpStack adds an explicit XHR timeout, since a plain browser XHR never fires on a silently-dead connection — which otherwise looks identical to a permanently hung upload. The widgets themselves (drag-drop zone, progress rows, pause/resume/retry controls, authenticated thumbnails) are consumed from ordinary forms through a FormX-File/FormInlineImage field type holding either just an id already on a record, or the richer object a fresh upload produced.

17.8 Consuming generated code

A real call site (an OTP-verification presenter) shows the pattern every generated action follows:

```
import { ClassicPassportOtpActionReq,
       useClassicPassportOtpAction }
  from "@fireback/selfservice/sdk/abac/
        ClassicPassportOtpAction";

const mutation = useClassicPassportOtpAction({});
const submit = (values) => {
  mutation.mutateAsync(new ClassicPassportOtpActionReq({
    ...values, value: state.value,
  }))
    .then(successful)
    .catch((error) =>
      form?.setErrors(mutationErrorsToFormik(error)));
};
```

The generated file supplies the mutation hook, a static class mirroring the emi action's JSON definition (URL, method, field descriptions), and the request/response DTO classes with private fields, typed accessors, and from/with/copyWith/clone helpers. Relations use MOne<T>/MCollection<T> wrapper types with explicit null-vs-undefined handling, so the backend can distinguish "explicitly cleared" from "untouched." All of it is regenerated by the emi compiler, never hand-edited — the same rule that governs every *.dyno.go file on the Go side (§16).

17.9 Supporting packages

Beyond the app-facing packages already covered, a handful of smaller ui/packages/* entries supply cross-cutting utilities consumed throughout the rest of the workspace: virtual-entity-manager (drag-and-drop reordering for generated entity lists, the Rank-column UI counterpart of the ordering field every dyno entity carries, §3.6), jsf (a JSONForms-like declarative form renderer, used where a form's shape is itself dynamic rather than a fixed hand-authored layout), overlay (the promise-based confirm-dialog/drawer/modal system every "are you sure?" and side-panel interaction in manage/selfservice is built on, covered in full in

§17.10 — notably one of the two packages, alongside resumable-uploader, that ships its own precompiled dist/ and therefore participates in the "must rebuild before it reaches the embedded binary" gotcha in §17.12), mobile-kit (dashboard widgets sized for a Capacitor-wrapped mobile build — one of the projectname build variants alongside local/bundle/manage), complexes (the frontend's own TString/locale-map types, mirroring the backend type of the same name in §3.4 so a translatable field round-trips through the generated SDK without a lossy conversion), and styles (shared theme CSS consumed by every app).

ui-core's hooks are what every screen actually reaches for day to day: useS resolves an i18n string key against the active locale (the frontend counterpart of a backend TString.Get), useRouter/useLocale wrap react-router-dom and the locale context respectively, and build-variables.tsx exposes the per-mode JSON injected by vite.config.ts (§17.3) as a typed object components can branch on (e.g. BUILD_VARIABLES.USE_HASH_ROUTER) without reaching into raw import.meta.env themselves.

17.10 The promise-based overlay system

Every modal, drawer, and confirm dialog in the UI — across manage, selfservice, and the demo gallery alike — is opened through @fireback/overlay's useOverlay() hook rather than through a component's own useState. The defining trait is that openModal/openDrawer/openPanel each return a *controller* whose .promise resolves with a DialogResult once the overlay closes:

```
export interface DialogResult<T> {
  data: T;
  // "resolved": user confirmed; "rejected": explicitly canceled;
  // "closed": passively dismissed (e.g. backdrop click / Escape).
  type: "closed" | "resolved" | "rejected";
}
```

Opening an overlay is a plain function call from anywhere — a click handler, an effect, another overlay's own body — with no JSX for the overlay itself sitting in the calling component's render tree, and no piece of boolean state (isOpen, selectedRow, …) to declare, thread through props, or forget to reset. ui/src/demo/ DemoModal.tsx (rendered live at the "Modal" demo screen) is a gallery of exactly this pattern; the examples below are lifted from it directly.

17.10.1 Why not useState?

The conventional React approach — a boolean (or selected-item) piece of state, an {isOpen && <MyModal … />} in the JSX, and an onConfirm/onCancel pair of callback props — works fine for exactly one modal, opened from exactly one place, with a result the opener doesn't need to chain off of. It stops working cleanly the moment any of that changes:

- **Getting a result back is inherently callback-shaped.** With state, "do X after the user confirms" means writing an onConfirm prop, storing whatever the modal collected in more state, and reading it back in an effect — state, a prop, and an effect just to move one value from the modal to its caller. With a promise, the caller .then()s it: *Example 3* below resolves a form modal's onSubmit with the typed value and immediately branches on it, all inline where the modal was opened, no extra state.
- **Multiple/stacked instances need an array of state, not a boolean.** *Example 4* fires four overlays (two drawers, two modals) from one click. The equivalent with isOpen booleans means either four separate flags or a hand-rolled stack data structure reimplementing what OverlayProvider already tracks in sheets.
- **Self-closing/self-updating overlays need imperative access from inside their own body.** *Example 5* closes itself from a setTimeout; *Example 7* conditionally blocks its own close via setOnBeforeClose based on "is this form dirty" state that lives *inside* the overlay. *Example 6* closes it from *outside*, two seconds after opening, using the controller's own close() — something a boolean flag can express, but only by re-deriving a ref/handle to flip it, which is exactly what the controller already is.
- **Live data pushed into an already-open overlay is a subscription, not a re-render trigger.** *Example 10* opens a drawer, then calls updateData() on its controller every 100ms from a setInterval outside the component, and .promise.finally() tears the interval down when the drawer closes for *any* reason — confirmed, canceled, or backdrop-dismissed. Wiring that with state means passing setters down as props

and manually clearing the interval in three different close-path branches instead of one finally.

- **Common dialogs shouldn't be reinvented per screen.** *Examples 8/9* call commonDialogs().confirmDrawer/ confirmModal — a shared "are you sure?" built once on top of openDrawer/openModal and reused everywhere a destructive action needs confirming, still awaited the same promise-based way as a fully custom overlay.

None of this is really about modals being "hard" with useState — it's that a modal's natural interface is *open it, then react to how it closed*, which is what a Promise already models, and async/await or .then() chains express directly. Modeling it as state instead means manually reconstructing promise-like bookkeeping (a result to store, a flag for "has it resolved yet", cleanup on every exit path) with plain data.

17.10.2 Examples

Opening a drawer is a single call, no local state at all:

```
const example1 = () => {
  return openDrawer(() => <div>Hi, this is opened in a drawer
      </div>);
};
```

Resolving a form modal with a typed value and branching on it inline:

```
const example3 = () => {
  openModal<string>(({ resolve }) => {
    const [value, setValue] = useState("");
    return (
      <form onSubmit={(e) => e.preventDefault()}>
        <FormText autoFocus value={value} onChange={(e)
            => setValue(e)} />
        <FormButton onClick={() => resolve(value)}>Okay</
            FormButton>
      </form>
    );
  }).promise.then(({ data }) => {
    if (data === "ali") return example1();
    alert(data);
  });
};
```

An overlay that blocks its own close while dirty, and one whose live data is pushed in from outside after it opened:

```
const example7 = () => {
  openModal(({ setOnBeforeClose }) => {
    const [dirty, setDirty] = useState(false);
    useEffect(() => {
      setOnBeforeClose?.(() => {
        if (!dirty) return true;
        return window.confirm("You have unsaved changes. Close
            anyway?");
      });
    }, [dirty]);
    return <input onChange={() => setDirty(true)} />;
  });
};

const example10 = () => {
  const { updateData, promise } = openDrawer(({ data }) => (
    <span>Params: {JSON.stringify(data)}</span>
  ));
  const id = setInterval(() => {
    updateData({ c: ++counter.current } as any);
  }, 100);
  promise.finally(() => clearInterval(id));
};
```

A shared confirm dialog, reused rather than hand-rolled per screen:

```
const example9 = () => {
  confirmModal({
    title: "Confirm",
    description: "Are you to confirm? You still can cancel",
    confirmLabel: "Confirm",
    cancelLabel: "Cancel",
  }).promise.then((result) => console.log(result));
};
```

17.10.3 Provider setup

The engine itself (@fireback/overlay's OverlayProvider) knows nothing about DOM markup — it only tracks the open-overlay stack (sheets) and exposes openOverlay/openModal/openDrawer/openPanel/dismissAll via context, taking a BaseModalWrapper and OverlayWrapper as props for how a "modal" or "drawer" should actually render. This is what lets the exact same promise-based API target a browser app (Bootstrap-based chrome, from the dom entry point) or, in principle, React Native (its own Modal/animated View wrappers) without changing a single call site. The dom entry point's DomOverlayProvider preconfigures the web chrome so an app only does:

```
import { DomOverlayProvider } from "@fireback/overlay/dom";

<DomOverlayProvider translations={myTranslations} locale={
    locale}>
  <App />
</DomOverlayProvider>
```

which is exactly how enterprise-shell's ApplicationOutlet.tsx mounts it once near the app root, above everything that later calls useOverlay(). OverlayProvider also wires the Escape key to close the top-of-stack overlay itself (a "panel"-type sheet, tracked but rendered by the host app rather than by the provider, requires a second Escape press — so an accidental tap doesn't discard in-progress work), and an optional uniqueIdentifier on any open* call makes a second attempt to open the same logical

overlay while one is already open a silent no-op, rather than stacking duplicates.

17.11 Notifications and the reactive convention

The in-app notification list is, deliberately, a plain GET-plus-react-query-refetch on a ten-second interval, not a websocket subscription. The reactive/websocket convention (method: "reactive", §12) is genuinely used elsewhere, though: the navbar search box opens a lazy websocket per keystroke (credentials passed as query params, since a WebSocket handshake cannot set an Authorization header) and streams results back as they match; and a single long-lived connection mounted once near the app root receives {cacheKey, name, payload} messages and calls queryClient.invalidateQueries(cacheKey) — i.e. the backend pushes cache-invalidation events and React Query is the actual delivery mechanism the rest of the UI observes, rather than individual components subscribing to the socket directly.

17.12 Build and embedding

Vite build modes produce plain static output in ui/dist. Two Go packages embed it directly:

```
// modules/interfaces/fireback-manage/index.go
package swiftpl
import "embed"
//go:embed *
var FirebackManageTmpl embed.FS
```

cmd/fireback/main.go registers both embedded trees as static-serving public folders under the running gin server (one mounted at /manage, the other at /selfservice), served through a shared embedding helper that also sets long-lived cache headers on hashed JS/CSS/image assets and no-store on index.html and the SPA fallback.

The Makefile targets that produce these embedded trees run the UI build, then copy ui/dist over the corresponding modules/interfaces/* directory (restoring only the Go index.go embed file via git checkout so the embed directive itself is never clobbered). A workspace build step runs first because a couple of ui/packages/* packages ship their own pre-compiled dist/, resolved through ordinary node_modules package resolution.

> *This is precisely the "pre-built embedded bundle, not hot-reloaded" gotcha already flagged in §2: a source change anywhere in* ui/ *has no effect on the running Go binary until someone re-runs the embed-refresh Makefile targets and rebuilds the server — nothing watches or auto-rebuilds this embed. A stale pre-compiled package* dist/ *has previously shipped an outdated CSS class name straight into the embedded binary and broken an end-to-end test, entirely silently, until the mismatch was noticed.*

Notable ui/ files and directories

- ui/src/index.tsx — the single Vite entry; conditionally imports one of three App.tsx.
- ui/vite.config.ts — multi-mode build config; build-variable injection; strips wasm assets outside wasm-demo.
- ui/packages/enterprise-shell/EssentialApp.tsx — the shared shell: query client, toast container, error boundary, sidebar/router mount.
- ui/packages/enterprise-shell/WithFireback.tsx — wires the auth provider, transport context, and the cache-invalidation websocket.
- ui/packages/auth-client/AuthenticationProvider.tsx — localStorage-backed session/workspace state, backend-agnostic.
- ui/packages/js-remote-ctx/ — generated transport primitives, regenerated wholesale, never hand-edited.
- ui/packages/selfservice/SelfServiceRoutes.tsx — the public vs. authenticated auth-flow route tables.
- ui/packages/manage/capabilities/, selfservice/roles/RolePermissionTree.tsx — capability/role tree editing.
- ui/packages/wallet/, walletpublic/ — finance module UI, each with its own generated SDK copy.
- ui/packages/resumable-uploader/ — the tus-based upload widgets.
- ui/packages/wasm-server/ — pglite/wasm bootstrap, imported only by the wasm-demo app.
- ui/src/demo/ — the component/module documentation gallery.
- modules/interfaces/{fireback-manage,selfservice}/index.go — the //go:embed * targets for the two embedded UI builds.

18 A Request, End to End

Every preceding section describes one layer in isolation. This section traces one concrete request through all of them, in order, as a capstone — showing how the pieces described separately in §3, §7, and §17 actually compose at runtime.

18.1 The scenario

A signed-in workspace admin opens the member-management screen in the React admin UI and changes another member's role. Concretely, this is the ChangeWorkspaceMemberRoleAction described in §9.3, triggered from RolePermissionTree.tsx (§17.5).

18.2 Step 1: the browser

The admin screen's form calls the generated mutation hook for this action — the same pattern shown for OTP verification in §17.8. React Query's mutateAsync serializes the request DTO to JSON and issues a POST through the shared fetchx transport layer (js-remote-ctx), attaching the caller's bearer token and Workspace-id header from AuthenticationProvider's current session state (§17.4).

18.3 Step 2: routing and the module system

The request lands on the gin server built by FirebackAppToGin (§3.1), which earlier wired every registered module's GinWebServerInitHooks — including abac's, since the action being called belongs to that module. Gin's router dispatches to the generated handler for ChangeWorkspaceMemberRoleAction, produced by the same action-level code generation described in §3.6: a typed request struct is decoded from the JSON body, and the generated handler calls into the hand-written implementation function.

18.4 Step 3: authorization

The implementation's first move is resolve.ResolveActionContext(request, securityModel) (§3.2, §9.6). Because this is a Gin request, resolveGinActionContext reads the Authorization, Workspace-id, and Role-id headers, builds an AuthContextDto, and calls resolve.WithAuthorizationPureDefault:

1. GetUserFromToken looks the bearer token up directly (§9.5), checks it hasn't expired, and resolves it to a user.
2. GetWorkspaceAndUserAccesses (§9.8) walks

that user's userWorkspace row for the workspace named in the header, follows every workspaceRole grant to its role, and flattens the granted capability keys.

3. MeetsAccessLevel checks the flattened capabilities against this action's declared ActionRequires (§9.2) — and, because this particular action also calls requireRealWorkspaceMembersCapability (§9.3), a second, stricter capability re-derivation runs as well, one that explicitly skips blank entries rather than trusting the same aggregation the first check used.

If either check fails, the handler returns a translated IError (§3.8) — NotEnoughPermission — and nothing past this point ever executes. GetSqlContext (§9.8) then turns the now-verified AuthResultDto into a fireback.QueryDSL (§3.7), stamping WorkspaceId/UserId from the *resolved* auth context — never from anything the request body claimed — which is what makes the request provably scoped to "this admin, in this workspace," independent of the permission check that already happened.

18.5 Step 4: the database

The action's own logic (§9.3) now runs: it deletes every existing workspaceRole row for the target membership and inserts the one new grant — a full replacement, not an append. Both statements go through gorm (§3.13) against whichever vendor's dialector the process was configured with (§3.3); on Postgres in production, on SQLite in a local dev checkout, or — in the wasm-demo build (§3.14, §17.3) — against pglite running in the very same browser tab that issued the request, through the identical gorm call, unaware anything about the transport underneath it is different.

18.6 Step 5: response and cache invalidation

The handler returns the updated membership wrapped in fireback's standard response envelope. Back in the browser, the mutation's onSuccess triggers React Query to refetch the member list, and — independently — the long-lived cache-invalidation websocket mounted by WithFireback.tsx (§17.11) may also deliver a push telling any *other* open tab watching the same workspace's member list to invalidate its own cached query, without that tab having initiated the change itself.

No layer in this trace knows about the others' internals. The generated HTTP handler doesn't know resolve *exists beyond calling one function;* resolve *doesn't know or care whether it was reached from Gin, a CLI command, or a websocket upgrade; gorm doesn't know whether its dialector is talking to a TCP socket or a JavaScript bridge; and the React hook consuming the response doesn't know whether the server that produced it was a real binary or the same page's own wasm build. That mutual ignorance — each layer depending only on a narrow, explicit contract with the next — is the actual payoff of everything described in the preceding eleven sections.*

19 Appendix: Glossary & Command Reference

This appendix consolidates the vocabulary and CLI surface introduced throughout the document into one place, for quick lookup rather than re-reading a whole section.

19.1 Glossary of core types

- TString — a locale → text map (§3.4), used everywhere a field needs to carry translated text (e.g. a capability's Name/ Description).
- MJson — an opaque JSON blob column, for fields whose shape doesn't warrant its own entity (e.g. role.capabilitiesListId).
- XDateTime / XDate — string-backed date(time) types with Jalali-calendar fallback and computed display metadata.
- PlainTime — the entity-reference documents' shorthand for complexes.Time, a nullable, presence-aware timestamp without locale awareness.
- IError / ferror.Error — the shared error envelope, carrying every locale's translation of a message plus field-level validation errors (§3.8).
- QueryDSL — the request-context struct threaded through every list/query action: offset or cursor, items per page, sort, filter string, column projection (§3.7).
- SecurityModel — the declaration of what an action requires to run: a set of permission keys, plus whether it's restricted to the root workspace (§3.2, §9.6).

- CompleteKey — the full dotted string identity of one permission (e.g. abac.bot.create), hand-written per file rather than generated by a shared helper.
- UniqueId — every entity's public string identifier, distinct from its internal auto-increment ID (§3.6).
- FirebackEmiAction / EmiActionBody / EmiField — the emi compiler types (§16) behind every top-level module action; entity fields share the same tags: {validate: required} validation convention.
- cliShort — the short mnemonic alias an emi action's generated CLI subcommand gets, wired as a cli.Command alias (§3.5).
- **ABAC** — attribute-based access control, this codebase's permission model (§7), implemented by modules/abac.
- **DSL** — domain-specific language; used here for both the QueryDSL filter string and the JSON-logic filter translator behind it (§3.12).
- **tus** — the resumable-upload HTTP protocol modules/storage implements (§10).
- **wal-g** — the Postgres backup/restore tool modules/backup embeds and orchestrates (§11).
- **pglite** — Postgres compiled to WebAssembly, backing the in-browser wasm-demo build (§3.14, §17.3).

19.2 Field type quick reference

- TString (map[string]string) — translated Name/Description fields.
- MJson (raw JSON blob) — role.capabilitiesListId, ad hoc structured data.
- XDateTime (RFC3339 string, Jalali fallback) — event timestamps with locale display.
- XDate — the date-only counterpart of XDateTime: birth dates, deadlines.
- complexes.Time ("PlainTime," nullable/presence-aware) — createdAt/updatedAt.
- TMoney (minor-units decimal, big.Int-backed) — wallet balances, ledger amounts.
- XFile — fields backed by modules/storage uploads.
- IError / ferror.Error (locale-keyed message + field errors) — every action's error return.

19.3 CLI command quick reference

Every entry below is a top-level app/fireback CLI command group; see the referenced section for full detail. Commands are opt-in — they only exist in a build whose main.go registers the owning module (§3.1).

fireback core

- init — first-time project setup wizard.
- config <key> get/set / config list — per-field and combined configuration (§3.3).
- db / dbdsn / ssl / sqllog — interactive setup wizards (§3.11).
- start — run the HTTP server.
- seeders — run every registered module's seed data.
- tasks / doctor / service — maintenance and diagnostics.
- public-folders — list embedded static-serving mounts (§17.12).
- about / version.

abac

- auth (alias authorize) — interactive/flagged end-to-end auth flow, CLI-only (§9.9).
- ws (alias workspace) — workspace/type/config/ invite/role/membership/public-join-key CRUD, plus scope, as, view, misc (including a totp simulator).
- user — user/token CRUD, accept-invite, user mock.
- bot — bot CRUD, refresh-bot-token.
- role — role CRUD.
- abac config — this module's own config get/set.
- abac internal — public-join-key, public-authentication, preference, user-profile,

pending-workspace-invite, capability (including capabilities-tree).

- parse — decode a token and print its resolved access.

backup, storage, eventbus, finance, internalstats

- backup push / verify / check / prune / restore / copy / dump / restore-dump (§11.3, §11.6).
- internalstats snapshot / watch — one-shot and live views of the same collector (§14.3).
- Storage, eventbus, and finance expose their CRUD/action surface through the generic emi-generated CLI pattern (§3.5) rather than hand-written command groups of their own, except where noted in each module's own section.

19.4 Recurring design principles

A handful of design choices reappear across otherwise unrelated modules in this codebase. Recognizing them makes unfamiliar code easier to read on first encounter, since the same shape tends to recur rather than each module inventing its own approach:

- **One mutation path per resource.** Wallet balances go through applyLedgerEntry and nothing else (§13.2); a bot's credential is only ever rotated by deleting the old token row and creating a new one (§9.4); a capability catalog entry is only ever inserted by CapabilityUpsertPermissionFn (§9.2). Concentrating every write to a piece of state behind one function is what makes invariants (a balance matching its ledger, a token never being edited in place) enforceable in one place instead of by convention across many call sites.
- **Idempotency keys over locks, where retries are expected.** Wallet mutations (§13.2) and download-job claims (§11.7) both treat a repeat request as a safe no-op rather than an error, specifically because the caller — a webhook sender, a flaky client — is expected to retry.
- **Defense in depth around a known, accepted gap.** The blank-capability bypass in MeetsCheck (§9.2) is not fixed at its source, because doing so would break tests relying on the exact gap; instead, the handful of actions where it actually matters add an independent, stricter check on top. Fix the highest-leverage call sites explicitly, rather than a blanket change with wide blast radius.
- **Opt-in composition over compile-time coupling.** Every module boundary in this document — internalstats accepting an Authorize callback instead of importing abac (§3.2), a payment gateway satisfying a four-method interface instead of being hard-coded (§13.5), a storage backend behind the Backend interface (§10.7) — is the same pattern: depend on a narrow interface, let main.go supply the concrete choice.
- **The generator is the source of truth, not the generated file.** Whether it's a *.dyno.go Go file (§3.6) or a generated TypeScript SDK file (§17.8), the rule is the same: edit the .emi.yml/Module3.yml definition and regenerate, never hand-edit the output (§16).

19.5 Where to look next

This document describes the codebase as of the date on its title page. For anything that has moved since — a renamed function, a new module, an updated emi schema — the authoritative source is always the repository itself: modules/*/*.emi.yml / *Module3.yml for what's declared, and the corresponding hand-written *.go files (never the generated *.dyno.go output) for how it's implemented. §16 covers the workflow for extending any of it.

19.6 Colophon

This document is set in Linux Libertine for body text, Linux Biolinum for headings, and Inconsolata for code, compiled with tectonic — a self-contained TEX engine that fetches exactly the packages a document declares, rather than requiring a full system-wide TEX distribution. The source lives at docs/latex/ in the repository root: one main.tex carrying the document class and styling, and one .tex file per section under sections/, each independently editable and \input-ed in order. Rebuilding after any change is a single command from the repository root:

```
make docs-pdf
```

which resolves to cd docs/latex && tectonic main.tex (Makefile, repository root) — no separate latexmk or system TEX installation required. make docs-pdf-clean removes any build cache tectonic

may have left behind; a plain tectonic invocation leaves none by default. The document is laid out double-sided (mirrored margins, alternating running headers, each top-level section opening on a fresh right-hand page) so a printed copy binds correctly — a detail that costs nothing when read on screen, since a PDF viewer simply shows consecutive pages either way.

19.7 Scope

This document covers the modules most projects touch first: fireback core, abac, storage, backup, eventbus, finance, internalstats, and the bundled React UI. Two modules mentioned only in passing deserve their own treatment in a future revision rather than a shallow pass here: modules/interfaces (the commercial server artifacts and embedded UI trees referenced in §17.12) and modules/abac/messaging (the email/SMS template and provider system introduced alongside notifications in §9.7, which has more real structure — regional content resolution, per-provider config shapes — than the notification flow alone shows). Likewise, the wasm-demo build's own React-side wiring (§17.3) is described at the level of *how it differs* from the ordinary app builds, not as a full tour of ui/packages/wasm-server in its own right. None of these gaps change anything already stated above; they are simply the next places to extend this document, following the same emi-first, read-the-actual-code approach used throughout it.

19.8 How this is tested

Two testing layers back the modules described in this document, at different altitudes. Go-level tests exercise business logic directly — §9.2 already mentions the roughly 180 tests under modules/abac/tests that pin down MeetsCheck's exact behavior, blank-capability gap included, closely enough that fixing the gap at its source would break them. End-to-end coverage lives under e2e/cypress, driving the real bundled UI against a real running server rather than mocking either side: support/setup.js performs the same non-interactive root-creation flow make devsetup automates (§2), and e2e/cypress/e2e/wasm-demo.cy.ts specifically drives the wasm and wasm-demo Makefile targets (§17.3) to confirm the in-browser build still boots end to end. A stale pre-compiled package dist/ feeding into the embedded UI has previously broken e2e/cypress/e2e/manage-regional-content.cy.ts silently (§17.12) — a concrete illustration of why the embed step is a real build boundary worth testing across, not an implementation detail safe to assume always matches source.

19.9 Building the server binary itself

Building this document is one command (§19.6); building the actual server binary the document describes has one environment prerequisite worth stating plainly, since it's easy to miss and produces a confusing failure otherwise: on a sufficiently recent Go toolchain, this repository (including modules/backup and the real fireback binary) only builds with GOEXPERIMENT=jsonv2 set. Without it, compilation fails in ways that don't obviously point at the missing experiment flag — worth checking first if a fresh checkout fails to build for reasons that don't match anything else in this document. This is a toolchain requirement of the checkout at the time of writing, not a property of the design described above; a later Go release may fold the experiment in by default, at which point this note becomes unnecessary.

19.10 Closing

That covers the ground this revision set out to: the core framework primitives every module builds on, the identity and permission layer those modules authorize against, five feature modules built on top of both, and the React frontend that ultimately puts all of it in front of a user — traced, in §18, as one request moving through every layer in order. Nothing here is meant to substitute for reading the code it describes; it exists to make that reading faster, by naming the shapes and conventions in advance so the first pass through an unfamiliar file spends less time reconstructing context the file itself never states outright.

19.11 A note on sourcing

Every claim in this document is sourced from the repository's own code and comments at the time of writing — file paths, function names, and the design rationale attached to them are drawn directly from modules/fireback, modules/abac, modules/storage, modules/backup, modules/eventbus, modules/finance, modules/internalstats, and ui/, not from external documentation or assumptions about how a similarly named system elsewhere might work. Where a comment in the source explicitly

records a past bug, a deliberate trade-off, or a rejected alternative, this document quotes or paraphrases that reasoning rather than restating only the current behavior — on the view that knowing *why* a piece of code looks the way it does is usually more useful to a new contributor than the shape of the code alone.

19.12 Feedback

If a section here turns out to be wrong, stale, or missing something a new contributor needed, the fix is the same as for any other bug in this repository: open an issue or a pull request against docs/latex/sections/ directly, citing the file and function that contradicts what's written. Because every claim above is meant to be traceable back to a specific place in the code (§19.11), a correction should be similarly traceable — pointing at the exact line that changed, rather than a general sense that a section feels outdated.

Appendix: The Emi Documentation

IMPORTED, UNMODIFIED, FROM THE EMI PROJECT

Heads up — this chapter isn't fireback's. The pages that follow are the **emi** compiler's own documentation, included here exactly as emi publishes it, so this appendix may look and read a little differently from the rest of the book. Fireback consumes emi *without any modifications* — every entity, action, DTO, and permission described in Section 16 is generated by this same, unforked compiler. For the latest version of this chapter on its own, see emi's GitHub Pages site and repository at https://github.com/torabian/emi.

Emi Compiler

One YAML API definition → type-safe Go, TypeScript, Swift, Kotlin, Python, Dart, C#, Java, PHP, C & C++ SDKs

Documentation **September 21, 2026** https://github.com/torabian/emi

Emi is a code generator: you describe your API once — dtos, entities, actions, config — in a single YAML file, and Emi compiles that one definition into working, type-safe code for multiple languages, so the backend and every client SDK are always generated from the same source of truth and never drift out of sync with each other.

Good fit for

- A Go/Gin backend paired with a TypeScript web app, mobile (Swift/Kotlin), and desktop clients from one definition.
- A game (Unreal Engine) or embedded device (ESP-IDF/Arduino) talking to that backend over real WebSockets.
- Publishing a multi-language **SDK** for your API — the way a payment provider, cloud platform, or any product with outside integrators has to hand every customer a client in *their* stack, not just yours, and keep all of those clients correct as the API evolves.
- Generating a **CLI** straight from the same action definitions the HTTP API exposes.
- Node.js **microservices**, including **NestJS** — the same generated JS/TS client that ships to the browser runs equally well server-side, so one client codegen covers both.
- Running codegen in-browser via **WASM** — no server round trip, same compiler as the CLI.
- Keeping a shared API contract in sync across a polyglot team without hand-syncing types.

Beyond basic DTO/action generation, Emi's goal is to cover the topics that usually get bolted on by hand afterwards: **reactive programming** (real WebSocket/SSE actions, typed on both ends), **nullability** (nullable-aware types across every target), **envelopes** (a consistent response-wrapper shape, e.g. Google's JSON style guide), **translations** (string-resource generation across languages), a first-class **CLI** (Golang actions double as `urfave/cli` commands for free), and **WASM** (the same compiler runs in the browser, no server round trip).

Quick start

1. Install Emi

```
go install github.com/torabian/emi/cmd/emi@latest
```

(or download a prebuilt binary from the releases page)

2. Create the sample yaml — save this as `user.emi.yml`:

```
complexes:
  - name: Money
    compiler: go
    location: github.com/torabian/emi/examples/fullstack/sdk/
        complexes
    namespace: complexes

dtos:
  - name: address
    fields:
      - name: city
        type: string
      - name: zip
        type: string?

  - name: user
    fields:
      - name: id
        type: string
      - name: email
        type: string
      - name: age
        type: int?
      - name: tags
        type: slice
        primitive: string
      - name: addresses
        type: collection
        target: AddressDto
      - name: balance
        type: complex
        complex: Money
```

`dtos` is a list of data-transfer object definitions — each one gets its own generated class/struct, one per target language. `fields` are plain, typed properties, and a trailing `?` (as in `int?`, `string?`, `collection?`) marks any field nullable, which every compiler renders the idiomatic way for that language (`emigo.Nullable[int]` in Go, `int?` in Swift/C#, `Optional[int]` in Python, ...). A few field shapes worth knowing:

- **Scalars** — `string`, `int`, `bool`, `float`, ...
- **`slice`** — a homogeneous list of a `primitive` scalar (here, `tags: []string`).
- **`array`** — a fixed-shape list where each item is itself an inline object (declared with its own nested `fields`).
- **`collection` / `one`** — a relation to another dto/entity via `target:` (`collection` is many, `one` is a single reference) — `addresses` here becomes `[]AddressDto` (or the equivalent per language).
- **`complex`** — an escape hatch to a hand-written type declared under `complexes:`, imported from a real language-native location instead of generated.

3. Compile it to multiple languages

```
emi go --path user.emi.yml --output ./out/go
emi js --path user.emi.yml --output ./out/js --tags typescript
emi swift --path user.emi.yml --output ./out/swift
emi python --path user.emi.yml --output ./out/python
```

Each command reruns the same `user.emi.yml` through a different target compiler and writes matching, type-safe code to its own output folder. You can also list every target once, under `targets:`, in the yaml itself and compile them all in one shot with `emi compile -path user.emi.yml`.

Emi YAML at a glance

A `.emi.yml` module is just a grouping of top-level blocks — this is the "table of contents" every compiler (Go, JS, Swift, ...) walks:

Block	What it's for
`namespace`	Where the module lives in the app tree (PHP-style), used as the client export path.
`dtos`	Plain data-transfer objects — shared shapes for request/response bodies.
`entities`	Database-backed structs; fields become Go struct fields and DB columns, and feed auto-synthesized CRUD actions.
`complexes`	Custom data types that don't fit the built-in field types.
`actions`	Controller-like units of behavior (HTTP and/or CLI), with typed `in`/`out` bodies.
`remotes`	Typed definitions of external HTTP services the module calls.
`config`	Typed server config, good for casting `.env` values.
`manifests`	Bundles of actions (include/exclude patterns) shippable as one unit (`go-client`, `go-gin`, `go-cli`, `go-wasm`).
`vsqls`	Hand-written SQL paired with a generated typed parameter struct.
`targets`	Self-contained compiler targets bundled with the module, for one-shot `emi compile`.
`templates`	Reusable dto/action shapes, never compiled on their own — only referenced (e.g. via `captures`).

Full schema: https://github.com/torabian/emi/blob/main/playground/public/emi-module-spec.json (works with the Red Hat YAML extension in VS Code for autocomplete/validation).

Language targets & features

Target	What you get
Golang	Full server + client: DTOs, `Gin` HTTP handlers, `urfave/cli` CLI, GORM entities, reactive/WebSocket support. The only target that generates a server.
JavaScript / TypeScript	Typed client SDK (vanilla or React + TanStack Query hooks), typed `fetch` layer, JSDoc typedefs for plain JS, reactive WebSocket/SSE hooks, NestJS decorators. Runs in the browser or Node.
Swift	`Codable` DTO structs, typed header structs, typed WebSocket client for reactive actions. HTTP actions are WIP.
Kotlin	Typed DTO/action client code. Reactive/WebSocket not yet supported.
Python	`dataclasses`-based DTOs, `httpx` HTTP client (sync or async), SSE for reactive actions.
Dart	DTO classes with hand-generated `toJson`/`fromJson`, `package:http` client, SSE for reactive actions.
C#	`System.Text.Json`-based DTOs, `HttpClient` transport, SSE via `IAsyncEnumerable<string>`.
Java	Jackson-based DTOs, `java.net.http.HttpClient` transport, SSE for reactive actions.
PHP	Reflection-based DTO hydration, `curl` HTTP transport, SSE for reactive actions.
C	Vendored `cJSON` (de)serialization, `libcurl` transport. No nullable value types without pointers (documented scope limit).
C++ (generic)	Portable ISO C++17 DTOs/actions for desktop, ESP-IDF, and Arduino; `IEmiHttpTransport` seam; real RFC 6455 WebSocket client for reactive actions.
C++ (Unreal Engine)	`USTRUCT`/`UPROPERTY` DTOs usable from Blueprint, Unreal's own JSON reflection, async HTTP via `FHttpModule`, real WebSocket via `IWebSocket`.

Every target except Golang is client-only — no server, no CLI, and no entity/GORM persistence is ever generated for them. Shared across all targets: nullable-aware types, nested object/array/map fields, `one`/`collection` relations, and enums.

> *"OpenAPI describes an API that already exists. gRPC replaces your transport with its own. Emi generates the API itself — backend and every client — as ordinary, runtime-free, idiomatic code you fully own."*

Contents

1. Why Emi Exists

Every team that ships an API ends up writing the same thing twice — once on the backend, where the route, the request body, and the response shape live, and again on the frontend, hand-copied into a `fetch` call, a model struct, a few `interface` declarations. Nobody decides to do this; it just happens, and the contract between backend and frontend isn't written down anywhere a compiler can see it. **Emi exists to make that contract a real, compilable thing**, and then generate every side of it, in idiomatic code, for every language a team ships.

Where Emi sits relative to what you've tried

- **vs. OpenAPI generators** — an OpenAPI spec is downstream of the real API, a second artifact that has to be kept in sync by hand or another generator. Emi inverts this: the Emi definition *is* the source, and your Go backend is generated *from* it, so the spec and the server can never disagree. Emi can still emit a standard OpenAPI 3 document and a Postman collection on demand (§13) — interoperability without making a hand-maintained spec the source of truth.
- **vs. gRPC/Protobuf** — gRPC solves a real version of this problem but asks for a binary wire protocol browsers can't speak natively, a runtime dependency in every client, and an all-in ecosystem. Emi generates **plain HTTP and plain JSON**, with **no Emi runtime** in the output — nothing to import, no library version to keep in lockstep. You can read every line Emi produces, hand-edit it, and eject at any time.

> *Emi's own one-sentence pitch: "OpenAPI describes an API that already exists. gRPC replaces your transport with its own. Emi generates the API itself — backend and every client — as ordinary, runtime-free, idiomatic code you fully own."*

How Emi generates code

The whole system rests on one idea: **the definition is target-agnostic**. An Emi definition (`.emi` YAML or JSON) describes *intent* — actions, DTOs, enums, fields, nullability, collections — and says nothing about Go, TypeScript, or Swift. A series of sub-compilers each turn that one definition into code for one target (the full fan-out is diagrammed in Fig. 1, §6.1). The CLI mirrors this directly:

```
emi go --path module.emi --output ./server
emi js --path module.emi --output ./web/sdk \
  --tags react,typescript
emi swift --path module.emi --output ./ios
emi kotlin --path module.emi --output ./android
emi openapi --path module.emi
```

`-tags` let one compiler shape its output without touching the definition: `-tags react` adds TanStack Query hooks plus `useSse`/`useWebSocketX`; `-tags nestjs` emits decorators that drop generated classes straight into Nest.js handlers; `-tags typescript` switches the JS compiler from runtime JS to `.ts`. None of them are mentioned in the definition — they're presentation choices made at compile time. Because the compiler is written in Go and also compiled to WebAssembly, the *exact same compiler* runs in the browser — that's what powers the live playground.

Why classes, not interfaces

Most codegen tools emit TypeScript `interfaces` — a compile-time promise, erased at runtime, so a server that starts sending `null` where it always sent a string throws in production while TypeScript shrugged at compile time. Emi's JS compiler emits real **classes** instead: private fields with typed getters/setters that coerce or reject bad values, **auto-initialization** of every non-nullable field (including nested objects, arrays and child DTOs, see §8), and real class instances handed back from the network (built through a `creatorFn`), not a raw JSON blob cast to a type. TypeScript checks types at compile time *and* the class enforces them at runtime — the safety Go programmers take for granted, in a language that historically had none of it.

2. The Module Definition

An Emi definition can be written as YAML or JSON — there is no special language of its own. At an abstract level it is just a definition; different sub-compilers (Go, JS, Swift, ...) produce different target code from the same root document, shaped by `-tags`. Every struct mentioned in this document lives in one place in the compiler's own source: `lib/core/Emi.go`.

A minimal module

```
name: sampleModule
actions:
  - name: getSinglePost
    url: https://jsonplaceholder.typicode.com/posts/:id
    cliName: get-single-post
    method: get
    description: Gets a specific post
    out:
      fields:
        - name: userId
          type: int64
        - name: id
          type: int64
        - name: title
          type: string
        - name: body
          type: string
```

Root-level module properties

name	lower camelCase; `Module.go`/`Module.dyno.go` are named from it
namespace	module's location in the app tree (PHP-style), used as the client export path
description	module purpose, used in codegen and generated docs
version	optional, useful across codegen phases
targets	compiler self-containment
enums	module-level enums, reusable across the rest of the definition
dtos	plain data shapes (Go structs) for request/response actions
entities	database-backed structures — fields become both struct fields and DB columns
complexes	custom data type definitions and their import location
actions	controller-like endpoints, HTTP and/or CLI (§3)
remotes	external HTTP services, made type-safe by declaring them here
config	server-side configuration manager
manifests	bundles of actions
vsqls	typed virtual SQL queries (see Query Predict, §15)
templates	reusable shape definitions
permissions	access-control tree the module contributes (§7)
events	facts the module can emit through the event system (§7)

dtos vs. entities

A `dto` is a plain data shape with no persistence of its own; an `entity` is database-backed — its fields become both a Go struct *and* migrated database columns (via gorm). Under the hood entities reuse the exact same field-rendering machinery dtos use — nested objects, CLI flags, JSON/YAML tags all work identically (§6.1 covers what's specific to entities: default `id`/`uniqueId` fields and gorm tags).

> *Emi definitions deliberately avoid target-language code, so the same YAML can be compiled into as many languages as the project grows to need — adding a tenth target is a new sub-compiler reading the same document, never a rewrite of the nine definitions that already exist.*

3. Actions

Actions are, in effect, functions: they take an input and return an output, corresponding to a "controller" in traditional API frameworks. Each action is an endpoint, but — unlike a typical controller — an action can also be called from contexts other than HTTP, for example the CLI, since `cliName` turns any action into a `urfave/cli` command for free.

```
name: sampleModule
actions:
  - name: getSinglePost
    url: https://jsonplaceholder.typicode.com/posts/:id
    cliName: get-single-post
    method: get
    out:
      fields:
        - {name: userId, type: int64}
        - {name: title, type: string}
  - name: sampleSse
    url: http://localhost:3000/stream
    method: get
    out:
      fields: [{name: message, type: string}]
  - name: webSocketOrgEcho
    url: "wss://echo.websocket.org/.ws"
    method: reactive
    in:
      fields: [{name: firstName, type: string}]
    out:
      fields: [{name: lastName, type: string}]
```

Action properties

name	required, camelCase, unique within the module
cliName	overrides the CLI command name (defaults to `name`)
cliShort	shorter alternative alongside `cliName`
url	HTTP route; omitted means the action is CLI-only
method	`post`/`patch`/`put`/`get`/`delete`/`reactive`
binaryType	websocket payload type, text by default
qs	type-safe query parameters, for both CLI and HTTP
description	used in specs and generated documentation
in / out	request/response body definitions

Only `name` is strictly required for an action to compile; `in`, `out`, `method` and `url` are simply the properties developers reach for most often. Calling a non-listed method falls back to HTTP with that non-standard verb. `reactive` is Emi's own addition on top of the HTTP-standard methods — it's what turns an action into a WebSocket channel instead of a request/response pair (§10), and a plain `get` whose handler streams chunked output is how an SSE action like `sampleSse` above is declared, with no separate keyword needed.

4. Intents: MCP Tool Generation

Where an action exposes a REST route or CLI command, an `intent` exposes the same kind of typed signature to an AI agent: a named, model-callable tool with a declared input and output shape, compiled into an MCP (Model Context Protocol) server's `tools/list` the way an action compiles into an HTTP route. Intents live in their own top-level `intents` list, sibling to `actions`.

```
name: gomcp
intents:
  - name: add
    title: Add two numbers
    description: Adds two numbers and returns their sum.
    annotations: {readOnlyHint: true, idempotentHint: true}
    in:
      fields:
```

```
      - {name: a, type: float64}
      - {name: b, type: float64}
  out:
    fields: [{name: sum, type: float64}]

- name: deleteInvoiceTool
  from: deleteInvoice
  title: Delete an invoice
  annotations: {destructiveHint: true}

actions:
- name: searchInvoices
  method: get
  url: /invoices/search
  intent: true
  in:
    fields: [{name: query, type: string}]
  out:
    fields: [{name: total, type: int}]
```

This one file shows all three ways to declare an intent. `add` is standalone: no backing action exists, since a pure math tool has no reason to be an HTTP route. `deleteInvoiceTool` is derived from an existing action via `from` — `description`/`in`/`out` are copied from `deleteInvoice` wherever the intent doesn't declare its own, so exposing an action as a tool needs no field duplication. `searchInvoices` needs no `intents` entry at all: `intent: true` on the action synthesizes one automatically, under the action's own name.

Intent properties

name	tool name shown to the model; defaults to `from`
from	name of an existing action to derive description/in/out from
title	human-readable title (MCP's `title` annotation)
description	sent verbatim to the model — write it for the model, not for a developer
in / out	request/response body, same shape as an action's
annotations	`readOnlyHint`, `destructiveHint`, `idempotentHint`, `openWorldHint`
permission	key from the module's permission tree gating visibility

Preprocessing resolves `intent: true` and `from` references before any compiler runs, and validates that a `dto` reference on an intent's `in`/`out` names a dto the module actually declares — a typo surfaces as a clear error at compile time instead of an "undefined type" failure deep inside generated Go.

What the Go compiler generates

`emi go` renders every intent into `Intents.go`: an `{Name}IntentArgs`/`{Name}IntentResult` struct pair for any intent with inline fields (a `dto` or `primitive` reference reuses the existing type instead, exactly as an action body would), through the very same struct generator dtos and actions use — so nested objects, arrays, enums, `complex` fields and CLI flag helpers all work identically. Alongside each struct pair sits an accessor carrying the tool's metadata:

```
func AddIntentMeta() struct {
    Name, Title, Description, Permission string
    ReadOnlyHint, DestructiveHint,
    IdempotentHint, OpenWorldHint bool
}
```

This is deliberately as far as generation goes: wiring `{Name}IntentMeta()` and the typed `Args`/`Result` structs into a real `*mcp.Server` is left to a hand-written file, the same division of labor a reactive action gets between its generated file and a developer-supplied `Impl` — `emi` has no opinion on which MCP SDK a project standardizes on, or on the actual tool implementation. `examples/go-mcp` shows the whole loop end to end: a two-tool math module compiled with `emi go`, served for real over MCP's stdio JSON-RPC transport by a few dozen lines of hand-written Go.

> *Because* `intents` *reuses the same field/struct/complex-type machinery as actions and dtos, an intent is never a second definition of a shape that already exists elsewhere in the module —* `from` *and* `dto` *both mean "point at the type that's already there."*

5. Complex Types

Instead of a primitive like `string` or `int`, a field can declare `complex: ClassName` — an escape hatch to a hand-written type living outside the generated file, imported from a real, language-native location rather than generated by Emi itself. A built-in type such as `Date` can be used directly; a user-defined type like `Money` has to be registered in the module's `complexes:` list, naming the class and where to import it from.

```
name: emiComplexFields
complexes:
  - name: Money
    location: ../some/directory/Money
actions:
  - name: sample
    method: post
    out:
      fields:
        - name: date
          complex: +Date
        - name: money
          complex: +Money
```

Prefixing the complex name with `+` (`+Date`, `+Money`) tells Emi to **instantiate** the value when parsing a response, instead of merely typing it. For the JS compiler this produces a real setter that coerces on assignment:

```
set money(value: Money) {
  if (value instanceof Money) {
    this.#money = value;
  } else {
    this.#money = new Money(value); // coerced, never trusted
         blindly
  }
}
```

`Date` is treated as a built-in complex and is never imported; `Money` is imported from the given `location` and instantiated on assignment. Every generated setter for a

complex field (`setDate`, `setMoney`, ...) returns `this`, so assignments chain.

> *On the Go side, a* `complex` *field used inside an* `entity` *has to pull its own weight: it must implement* `database/sql/driver.Valuer` *and* `sql.Scanner` *itself if it needs to persist as anything other than gorm's default guess — the generator can't know how to store an arbitrary hand-written type (§6.2). Writing the JS/TS-side class is covered in §8.3.*

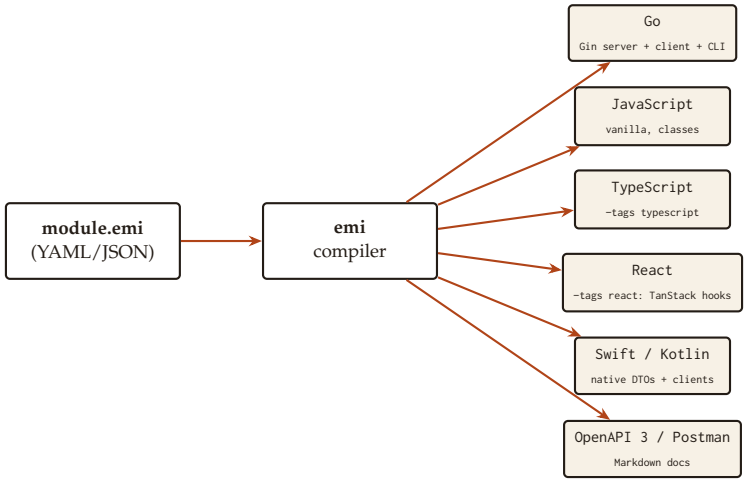

Fig. 1 — The compilation pipeline. One target-agnostic definition, many sub-compilers, each reading the same document. The CLI mirrors this diagram directly — `emi go`, `emi js`, `emi swift`, `emi kotlin`, `emi openapi` — and `-tags` (`react`, `nestjs`, `typescript`) layer presentation choices onto a target's output without the definition ever mentioning them. Because the compiler itself is written in Go and also builds to `GOOS=js GOARCH=wasm`, the exact same compiler that produces this fan-out also runs client-side in a browser tab — no server, no install (§16, and the live playground).

6. Go Compiler Features

The Go compiler is the only Emi target that generates a full server, not just a client for one — so this section tours what it adds on top of the shared DTO/action codegen already covered above: database-backed entities, typed relations with real migratable gorm associations, and the create/update actions generated for every entity. Each part below shows a small sample of the Go code the compiler actually emits.

Entities

An entity is a database-backed structure: its fields become both a Go struct and database columns, migrated with gorm. Declared in a module's `entities:` list, right next to `dtos:`/`actions:`:

```
entities:
  - name: entity1
    table: entity1_table
    description: Exercises every supported field type.
    fields:
      - name: title
        type: string
```

`name` becomes the Go struct name (always upper-cased with `Entity` appended — `Entity1Entity`); `table` overrides gorm's default pluralized/snake_cased table name.

Two fields every entity gets for free

```
type Entity1Entity struct {
  Id int64 `gorm:"primaryKey;autoIncrement" json:"-"`
  UniqueId string `gorm:"type:uuid;default:gen_random_uuid();
       unique"
                  json:"uniqueId"`
  Title string `json:"title"`
}
```

`Id` is the real gorm primary key — a plain auto-incrementing integer, never exposed over JSON/CLI, so joins and foreign keys can use a small integer instead of a UUID (smaller indexes, faster joins). **`UniqueId`** is the public identifier every API/CLI surface actually works with: a UUID generated natively by Postgres itself (`gen_random_uuid()`, built into Postgres 13+) via a column default, not application code. Both are prepended once, in `EntityDefaultFields` (`lib/golang/go-entity-default-fields.go`), before the struct generator, the update-input builder, and the gorm-tag pass all run — every consumer sees them as ordinary declared fields.

Field types on gorm

Field type	Gorm tag
scalar (`string`, `int`, ...)	none — gorm maps it directly
`enum`/`enum?`	none — already a plain `string`/`Nullable[string]`
`object`/`object?`	`embedded` (inlined as columns)
`map`, `slice` (non-nullable)	`serializer:json`
`map?`, `slice?`	none — `Nullable[T]` already implements `Value()`/`Scan()`
`complex`	none — the type implements gorm's interfaces itself
`array`/`collection`/`one`	see relations, §6.3

An explicit `tags: { gorm: ... }` on a field always wins over these defaults. A `complex` field used on an entity must implement `driver.Valuer`/`sql.Scanner` itself — the generator has no way to guess how to persist a hand-written type.

Every entity file ends with a small, separate template (`go-entity.tpl`) whose output is appended *after* the generated struct — currently a `TableName()` override wired to `table:`, and the place to extend if a project wants something appended to every generated entity file.

Browsing: JSON Logic filters

An entity with browsing enabled also gets a generated `{Entity}BrowseFn` and a `GET /entity1/browse` action, exposing five query params: `filter`, `sort`, `startIndex`, `itemsPerPage`, `cursor`. `filter` is a JSON Logic expression, transpiled straight to SQL by `emigorm.ApplyQueryFilter` — never string-concatenated, and never a filter language Emi invented itself:

```
GET /entity1/browse?filter={"==":[{"var":"title"},"hello"]}
GET /entity1/browse?filter={"and":[
  {">":[{"var":"age"},18]},
  {"contains":[{"var":"title"},"draft"]}
]}
```

Every standard JSON Logic operator works (`==`, `!=`, `>`, `<`, `and`, `or`, `in`, ...) plus one Emi adds itself: `contains` — a case-insensitive substring match (`::text ILIKE '%...%'`), the one operator plain JSON Logic has no native equivalent for.

Sort and paging are the plain, non-JSON-Logic half: `sort=createdAt desc` (empty defaults to `id asc`, needed for keyset paging to stay correct), and either offset paging (`startIndex`/`itemsPerPage`) or cursor paging (`cursor`, echoing the response's own cursor field back on the next call — `cursor=id(42)`).

> `filter` *is the only* user-*controlled half of a Browse query. A handler's own tenant/workspace isolation goes through the separate* `emigorm.ApplyQueryScope` *(a raw SQL fragment the handler itself writes, applied unconditionally) —* `filter` *has no way to see or override it.*

Writing a complex type

A `complex: Money` field (§5) needs a real Go type at the declared `location`/`namespace` — nothing more, to just carry the value through a DTO or action. Persisting it on an `entity` is a separate, opt-in step: gorm has no idea how to store an arbitrary hand-written struct, so the type has to say so itself.

Nullability: no wrapper, no pointer

A `complex` field is the one place Go's nullable-field convention doesn't apply. Every other nullable field type gets wrapped in `emigo.Nullable[T]` (§6.1.2); a complex field never does, whether the field is declared `complex` or (the JS-only variant) `complex?` — both compile to the exact same plain `Money` value in Go, because a complex value has no wire-level nullability of its own as far as the Go compiler is concerned. `complex?` exists purely so the JS/TS compiler can tell "always instantiate" apart from "allow `undefined`/`null` on the setter"; every other backend, Go included, treats the two identically.

The one place a complex field *does* become a pointer is inside the generated `{Entity}UpdateInput` struct (§6.4): since there's no `Nullable[T]` counterpart for a hand-written type, the update-input builder falls back to a plain `*Money`, where `nil` means "untouched" — the same "was this field mentioned at all" signal every other field gets from `Nullable[T]`'s own `IsSet()`, just spelled with a raw pointer instead of a generic wrapper.

Persisting to one column

Implementing `database/sql/driver.Valuer` and `sql.Scanner` stores the whole value as a single column:

```
type Money struct {
  Cents int64
}

func (m Money) Value() (driver.Value, error) {
  return m.Cents, nil
}

func (m *Money) Scan(value interface{}) error {
  switch v := value.(type) {
  case int64:
    m.Cents = v
  case []byte:
    cents, err := strconv.ParseInt(string(v), 10, 64)
    if err != nil {
      return err
    }
    m.Cents = cents
  }
  return nil
}
```

A real complex type is expected to implement whatever interfaces its own storage needs — `MarshalText`/`UnmarshalText` too, if it should also cast cleanly from a CLI string argument, the same way fireback's own complex types (e.g. `XDate`) do.

Persisting across multiple columns

Valuer/Scanner is a single-column round trip; not every complex type fits one column. Skip both interfaces and gorm falls back to walking the struct like it does for an `object/object?` field (§6.1.2) — each exported field becomes its own column:

```
type Address struct {
  Street string `gorm:"column:addr_street"`
  City string `gorm:"column:addr_city"`
  Zip string `gorm:"column:addr_zip"`
}
```

No `Value()/Scan()` at all here — the struct is plain data, and gorm's own reflection spreads it across `addr_street`, `addr_city`, `addr_zip` exactly as it would for any embedded struct. Pick single-column when the type has one natural serialized form (money as cents, a date as text); pick multi-column when it's naturally shaped as several fields side by side.

Relations: array, collection, one

A dto's `one/collection` field just needs a Go type to hold data. An entity's relation field needs that *and* a real, migratable gorm association — and the two shapes are incompatible in a way that isn't obvious up front. `array/collection/ one` normally render as `emigo.Array[T]/ Collection[T]/One[T]` — PATCH-payload wrapper types carrying an `Operation` (`replace/append/ select`) — the right shape for an API, but not one gorm can migrate (verified directly against `gorm.io/gorm/schema.Parse`).

So an entity keeps the DTO field as-is (still the only public JSON/CLI surface) tagged `gorm:"-"`, and generates a hidden sibling field that *is* a real gorm association:

```
Items emigo.Array[Entity1EntityItems] `gorm:"-" json:"items"`
Owner emigo.One[Entity2Entity] `gorm:"-" json:"owner"`

ItemsRow []*Entity1EntityItems `gorm:"foreignKey:LinkerId;
    references:Id;
                              constraint:OnDelete:CASCADE" json
                                  :"-"`
OwnerId int64 `gorm:"index" json:"-"`
OwnerRow *Entity2Entity `gorm:"foreignKey:OwnerId;references:Id
    " json:"-"`
```

The `{field}Row/{field}Id` siblings are purely a persistence concern (`json:"-"`); whatever writes to the database (§6.4) keeps the DTO field and its sibling in sync.

The three relation kinds

array — has-many, owned composition. Nested `fields:` become a child struct with its own `Id/UniqueId` plus a `LinkerId` FK back to the parent.
collection — many-to-many, backed by a join table named `{entity}_{field}` (e.g. `entity1_items3`).
one — belongs-to: a foreign-key column on *this* entity, named `{field}Id`, referencing the target's `Id`.

Why `Id` and not `UniqueId`: every `foreignKey/references` tag joins on the small integer `Id`, keeping indexes smaller and joins faster, even though `UniqueId` is the only identity an API caller ever actually has. Since gorm's `Save()` decides insert-vs-update purely from whether `Id` is already populated, and a caller only ever knows `UniqueId`, the reconcile helpers (§6.4) resolve an incoming item's real `Id` by matching `UniqueId` first.

Relations nested inside object containers

A relation doesn't have to sit directly on the entity — it can live inside an `object` (or `object?`) field, any depth deep. Gorm's own embedding correctly walks an arbitrarily deep `embedded` chain and still recognizes the association, keyed by its bare field name rather than the full nested path (verified against `gorm.io/gorm/schema.Parse` and a real database run) — provided the nested child struct is named with the same accumulating prefix the struct generator uses for the container itself, and `{Entity}CreateFn/UpdateFn` recurse into `object/object?` containers to find relations at any depth. `object?` specifically needs its own fix: it renders as `emigo.Nullable[T]`, and gorm's embedding only ever sees the wrapper's own unexported fields, never `T`'s — so it gets the same `{field}Row *T` hidden-sibling treatment a relation does, even for a field with no relation in it at all.

Known limitations: a relation declared on an array item itself (rather than on the entity or one of its object containers) isn't picked up — each array item is still persisted as one flat unit via `Save()`. Cross-module targets (`module:` on a relation field) aren't accounted for in the FK/table naming yet.

Create & Update Actions

Every entity gets two generated functions — `{Entity}CreateFn` and `{Entity}UpdateFn` — plus a small actions bundle wiring them up as the defaults, mirroring the pattern fireback uses.

Request body formats

Every generated Go action — Create/Update included — reads its body through `emigo.BindGinRequestBody`, which picks the decoder from the request's own `Content-Type` rather than assuming JSON:

- `application/json` — the default, and what every generated client sends.
- `application/yaml/x-yaml/text/yaml` — the same struct, from YAML.
- `application/xml/text/xml`.
- `multipart/form-data` — each non-file field round-trips through JSON into the struct; each uploaded file becomes a `BindMultipartFile` (`filename/mime/ data`), unless the host app registers its own `ConvertMultipartFile` (fireback, for instance, turns one into an `XFile`).
- `application/x-www-form-urlencoded` — a plain HTML `<form method="post">` postback works unmodified, no JavaScript or fetch call required: the browser's own form submission already sends this content type, and every field lands in the same generated struct a JSON body would.

Only `POST/PATCH/PUT` read a body at all — a no-op for every other method, matching plain HTTP. The response side mirrors this: `RenderGinResult` content-negotiates off the request's `Accept` header (JSON, YAML, TOML, or CSV; JSON

when unset or unrecognized), so a caller gets the format it asked for without the handler itself knowing or caring which one that is.

The update input: optional, all the way down

A struct-based `gorm.Updates()` call silently drops zero-valued fields, so it can't tell "the caller didn't mention this field" apart from "the caller explicitly set it to zero." Every entity therefore gets a second generated type, `{Entity}UpdateInput`, where *every* field — even a plain, always-present scalar on the entity itself — is wrapped optional:

```
type Entity1EntityUpdateInput struct {
  Title emigo.Nullable[string] `json:"title"`
  Items emigo.ArrayNullable[Entity1EntityItems] `json:"items"`
  Owner emigo.OneNullable[Entity2Entity] `json:"owner"`
  Complex1 *Money `json:"complex1"`
}
```

`id`/`uniqueId` are left out entirely — immutable identity, not something an update patches. A `complex` field (no `?` counterpart exists) becomes a plain pointer instead; `nil` means untouched. Critically, `array`/`collection`/`one` fields reuse the *real* row/target type directly, not a second, separately-optional clone — only whether the field itself was touched (`IsSet()`) is optional; an item included in a replace/append batch still has to be given in full, exactly like Create, because the reconcile helpers persist each item with a plain `Save()` that writes every field.

The reconcile helpers (`emigorm`)

Gorm's own `Association().Replace()` was verified **unreliable** for a has-many `array` relation: an incoming item whose primary key already exists silently keeps its *old* content, and items dropped from the list are never deleted — just orphaned. So `github.com/torabian/emi/emigorm` (kept out of the wasm-safe `emigo` core since it depends on `gorm.io/gorm`) provides three reconcile functions instead:

One reconciler per relation kind

ReconcileHasMany (`array`) — diffs existing children against the incoming list by `UniqueId`, hard-deletes any row missing from a `"replace"` batch, `Save()`s every incoming item. `"append"` skips the diff/delete step.
ReconcileManyToMany (`collection`) — gorm's Association API *is* reliable here, so this upserts inline target values first, then delegates to `Association().Replace()`/`.Append()`.
ReconcileOne (`one`) — `"select"` resolves an existing target's `Id` from its `UniqueId` and leaves it untouched; anything else upserts the inline value.

All three resolve an incoming item's real `Id` by matching `UniqueId` against an existing row *before* calling `Save()` — otherwise every "update" would silently insert a duplicate.

Testing

Every relation is verified two ways: **structurally**, with no database at all, via `gorm.io/gorm/schema.Parse` (confirms every `has_many`/`many_to_many`/`belongs_to` resolves correctly in milliseconds); and **end to end against real Postgres** (`examples/emi-entity/sdk/*_test.go`) — cascading create, partial update, array replace-with-orphan-deletion, collection append, and one re-select are all covered. Since `UniqueId`'s column default is Postgres-specific, the live tests need a real Postgres instance (`make entity-db-up`); MySQL is wired up too but expected to fail on the same `gen_random_uuid()` default — an accepted tradeoff, not something actively kept green.

7. Permissions & Events

Permissions: a compiled tree

A module can declare a `permissions:` tree — nested access control permissions the module contributes. Each node has a `key` (its slug) and a `name` (used to name the generated identifier, e.g. `name: managePosts` → `ManagePostsPermission`). The full identifier is either set explicitly via `fullKey:` or computed as `{parent.fullKey}.{key}` during preprocessing; a node that groups children and left its key to be derived gets a trailing `.*` (e.g. `post.*`), signaling it covers itself and everything below.

```
permissions:
  - key: post
    name: managePosts
    children:
      - key: create
        name: createPost
      - key: publish
        name: publishPost
```

Compiling this (Go target) produces one exported var **per root permission** — not one var wrapping the whole forest, and no separate flat constant per node either. Each var is a fully typed, nested struct: every node embeds `emigo.Permission` (promoting its own `Key`) and has one field per child, so an editor autocompletes straight down it with zero risk of a typo'd key only failing at runtime:

```
var ManagePostsPermission = struct {
  emigo.Permission
  CreatePost emigo.Permission
  PublishPost emigo.Permission
}{ /* ... */ }

var ManagePostsPermissionList = []emigo.Permission{
  ManagePostsPermission.Permission,
  ManagePostsPermission.CreatePost,
  ManagePostsPermission.PublishPost,
}
```

`<Name>PermissionList` sits alongside every root, built by referencing the root var's own fields rather than re-declaring separate literals, so it can never drift out of sync with it — one list per root, not one combined list a caller has to filter back down. The JS/TS compiler mirrors this shape as a `const ... as const` object (or `Object.freeze(...)` without `-tags typescript`) instead of a Go struct, with the same one-const-per-root, `<Name>PermissionList` convention.

Events: flat facts, typed payloads

A module can declare an `events:` list — flat, unlike `permissions:`, since an event is a single fact ("a post got published"), not a tree. Each event has a `key` (its wire name), optional localized `name`/`description`, a `payload`, and

permissions describing who's allowed to know it happened — a list of with blocks, OR'd together, each entry's with list AND'd ([{with: [A,B]}, {with: [C,D]}] means (A and B) or (C and D)).

```
events:
  - key: postPublished
    permissions:
      - with: ["post.publish"]
      - with: ["post.*"]
    payload:
      fields:
        - {name: postId, type: string}
  - key: commentAdded
    payload:
      dto: CommentDto
  - key: postArchived
```

Three events, three payload shapes: postPublished declares fields inline, compiled into a dedicated PostPublishedEventPayload struct exactly like an action's request body, complete with CLI flags; commentAdded points dto: at an existing dto, so its payload type is a plain alias, not a new struct; postArchived declares no payload at all, so its payload type aliases straight to interface{}. One exported var is generated per event (PascalCase, trailing Event), always paired with a <EventName>Payload type and a New<EventName>(payload) constructor, and every var is collected into AllEventsList for anything that wants to walk the whole catalog. The compiler rejects the module outright if two differently spelled keys (postPublished vs. post_published) would normalize to the same generated identifier, rather than letting one silently shadow the other.

8. The JavaScript/TypeScript Compiler

JavaScript gets special focus in Emi, because nearly everything eventually has to be consumed by a JS environment. The JS output is real, runnable JavaScript — no transpile step required — while -tags typescript produces high-coverage .ts with, Emi's docs claim, some of the highest-quality TypeScript output among competing generators. -tags react layers on TanStack React Query hooks plus an internal useSse and useWebSocketX; -tags nestjs emits decorators that make request, headers and query params portable straight into Nest.js handlers with no extra code.

Auto-init: every field guaranteed to exist

When an action returns a class instance, *every* non-nullable field — including nested objects, arrays, and child DTOs — is guaranteed to exist. Writing this by hand is a hassle and easy to get subtly wrong; Emi generates it from the definition alone:

```
fields:
  - name: object1
    type: object
    fields:
      - name: contacts
        type: array
        fields:
          - name: email
            type: string
            default: emi-compiler@emi-compiler.com
          - name: phone
            type: string?
```

produces a class whose object1 field is *always* an instance of a nested AutoInitClassDto.Object1 class, never undefined — the code stops sprinkling ?., ??, and typeof x === 'string' defensively, because the class already did the guarding.

Collection and One: replace, append, select

array, collection, and one fields compile to MArray<T>, MCollection<T>, and MOne<T> — wrapper classes from sdk/common/operators that mirror Go's own emigo.Array[T]/Collection[T]/One[T] (§6.3): the same PATCH-payload semantics, JS side. A generated setter accepts either a real wrapper instance or a plain array/object and coerces it, so most call sites never construct a wrapper by hand:

```
user.collectionField = [new User({ name: "Ada" })]; // replace (
    the usual case)
user.collectionField = MCollection.append([newUser]); // append
    to the existing set instead
user.oneField = MOne.select(existingUser.uniqueId); // resolve
    by selector, leave the target alone
```

MOne.select(selector) mirrors ReconcileOne's "select" operation on the Go side (§6.4): it resolves an existing target by selector without touching its fields, instead of upserting an inline value. Every wrapper shares the same accessors — .get() returns the plain value/array underneath, .len() (array/collection) counts it, and .isReplace()/.isAppend()/.isSelector() report which operation is active.

Why toJSON() matters here

Each wrapper defines its own toJSON(), so JSON.stringify() on a DTO that contains one never needs special handling: a "replace" serializes as a bare array/value (the implicit, most common case on the wire); "append" and "select" serialize as a tagged object instead — {__operation, items} or {__operation, __selector} — so the server-side reader knows which operation to apply before it ever looks at the payload shape.

Writing a complex class

A plain (non-instantiated) complex: Money field needs nothing more than a real class at the declared location — Emi imports it and types the field with it, never generates it. The minimal contract is just a constructor that accepts whatever the field's raw value looks like:

```
export class Money {
  constructor(data: unknown) {}
}
```

When the field uses the + prefix (complex: +Money, §5) the generated setter calls new Money(value) on anything that isn't already instanceof Money, so the constructor is where a real implementation normalizes whatever shape

the server actually sent (a number, a string, a nested object) into the class's own fields:

```
export class Money {
  #cents: number;
  constructor(data: unknown) {
    this.#cents = typeof data === "number" ? data : (data as any
        )?.cents ?? 0;
  }
  toJSON() { return this.#cents; }
}
```

Nothing else is required — no interface to implement, no registration beyond the `complexes:` entry. The Go side of the same type has its own, separate contract (§6.2): the two are unrelated hand-written files that happen to share a name in the `complexes:` list.

Ergonomics every generated class ships with

- `from()` — full object constructor; `with()` — partial, deeply-typed via `PartialDeep`.
- `copyWith()`, `clone()`, a correct `toJSON()` that sees the private fields (plain `JSON.stringify` otherwise can't — the fields are genuinely private).
- A static `Fields` map (e.g. `Fields.date === "date"`) for safe, refactor-proof path access instead of a hardcoded string literal.

The Headers class

HTTP request/response headers are a key/value definition; Emi generates a class extending the standard `Headers` with typed `getX()`/`setX()` accessors that coerce values on the way out:

```
- {name: accept-language, type: string}
- {name: cache-time, type: int64}
```

```
export class SampleHeaders extends Headers {
  getAcceptLanguage() { return this.#getTyped('accept-language',
       'string|null'); }
  setCacheTime(value: number|null) {
    if (value !== null) this.set('cache-time', value);
    return this;
  }
}
```

`.get()` on the plain `Headers` base still returns the raw string — the typed accessors are additive, not a replacement.

> *Header/query-param typing at this level — including* nested *query params rendered as typed accessors — is a feature Emi's own docs single out as unique among the generators it's compared against.*

React Query: hooks and the provider

`-tags react` generates one `use{Action}Query` hook per GET-style action (and a matching mutation hook for write actions), each one a thin wrapper around TanStack Query's own `useQuery`. The query key is derived from the action's own URL-building function plus its query-string options, so two components calling the same action with the same `params`/`qs` share one cache entry automatically — nothing to key by hand:

```
const { data, isLoading, response } = useGetSinglePostQuery({
  params: { id: postId },
});
```

Every generated hook reads its HTTP config — base URL, default headers, request/response interceptors — from React context via `useFetchxContext()`, not from arguments the caller passes in. That context has to be set up once, near the app root, alongside TanStack's own `QueryClientProvider` (a plain TanStack Query prerequisite, not something Emi generates):

```
import { QueryClient, QueryClientProvider } from "@tanstack/
    react-query";
import { FetchxProvider } from "./sdk/react/useFetchx";
import { FetchxContext } from "./sdk/common/fetchx";

const queryClient = new QueryClient();
const fetchxContext = new FetchxContext(
  "https://api.example.com", // baseUrl, prefixed onto every
      relative request
  { "x-app": "web" }, // defaultHeaders
);

export function AppProviders({ children }) {
  return (
    <QueryClientProvider client={queryClient}>
      <FetchxProvider value={fetchxContext}>
        {children}
      </FetchxProvider>
    </QueryClientProvider>
  );
}
```

`FetchxContext` also takes a `requestInterceptor` (e.g. to attach an auth token before every request) and a `responseInterceptor` (e.g. centralized 401 handling) — both run for every generated hook and every plain, non-React call through `fetchx()` alike, since hooks are just `useQuery` sitting on top of the same context.

9. JSON Schema on the JavaScript Side

Every JS/TS dto already has a field-plan tree the compiler walks to render a form (§8); `lib/formgen` walks that same tree a second way, into a renderer-agnostic JSON Schema instead of a bespoke component. There are two ways to get one out.

Two ways to get a schema

Per-dto, standalone — `emi js:rjsf` takes a single `emi: dto` definition and writes two files: the schema itself, and a thin wrapper component around react-jsonschema-form (RJSF) that renders it:

```
emi js:rjsf --path address.emi.yml --output ./out
# ./out/AddressDto.schema.json
# ./out/AddressDtoForm.tsx
```

Embedded on the class — `-tags json-schema` on a full module compile instead attaches every generated dto/action-request/ action-response class's schema directly to itself, as `static JsonSchema = {...}` (off by default — it's extra bytes most consumers don't need):

```
emi js --path module.emi.yml --output ./out --tags json-schema,
    typescript
```

Both read the exact same `BuildJSONSchema` tree walk (`lib/formgen/jsonschema.go`), so the shape of the schema itself never depends on which of the two you use — only where it ends up.

A minimal example

```
name: address
fields:
  - name: street
    type: string
    description: Street and number
  - name: unit
    type: string?
  - name: kind
    type: enum
    enum:
      - key: home
        description: Home address
      - key: work
  - name: manager
    type: one
    target: employee
```

compiles (`emi js:rjsf`) to:

```
{
  "type": "object",
  "title": "AddressDto",
  "properties": {
    "street": { "type": "string", "title": "Street", "
        description": "Street and number" },
    "unit": { "type": "string", "title": "Unit" },
    "kind": { "type": "string", "title": "Kind", "oneOf": [
      { "const": "home", "title": "Home address" },
      { "const": "work", "title": "work" }
    ]},
    "manager": { "title": "Manager" }
  },
  "required": ["street", "kind", "manager"]
}
```

Two things worth noticing

`unit` is missing from `required` — the trailing `?` is the only thing that removes a field from it, independent of the field's own type or widget. And `manager` — a `one` relation, so genuinely unconstrained (`{title: "Manager"}`, no `type`) — is still *in* `required`, because it isn't nullable: `required` tracks nullability alone, not whether the schema actually constrains the value. RJSF only checks the key is present, so an empty object still satisfies it.

An `enum` option's human label comes from its `description` (falling back to the raw key, as `work` does above); `one`/`collection` relation fields and `any`/`complex` fields both resolve to an unconstrained `{}` on purpose — the schema compiler was never given the target entity's own fields (a relation) or has no fixed shape to begin with (`any`), so it leaves the value alone rather than guessing, deliberately mirroring the same "leave it to a custom field on the renderer side" choice §8.3 makes for a hand-written complex type.

Using it in the front end

The RJSF wrapper is a controlled component — `value`/`onChange`, the same convention every other generated form component in Emi uses — so a consuming app drops it in with zero manual field wiring, validation included:

```
import AddressDtoForm from "./out/AddressDtoForm";

function EditAddress() {
  const [value, setValue] = useState<Partial<AddressDto>>({});
  return <AddressDtoForm value={value} onChange={setValue} />;
}
```

Under the hood that wrapper is nothing more than the schema plus RJSF's own `<Form>` and `@rjsf/validator-ajv8` — swap either import and the same generated `.schema.json` still works, since nothing about it is RJSF-specific.

The embedded `static JsonSchema` form exists for the opposite case: an app that isn't using RJSF at all still gets a real JSON Schema for free on every generated class, to hand to a different schema-driven renderer, validate a payload with a bare `ajv` instance before a submit, or drive a headless dynamic-form engine — all without a second compile step or a hand-maintained schema that can drift from the dto.

> *Both paths are opt-in and additive: no code Emi already generates for a dto changes shape because a schema exists alongside it, and skipping both (the default) costs nothing — no `formgen` import ends up in the output at all.*

10. Reactive: WebSocket & SSE

Marking an action `method: reactive` tells Emi the endpoint is a WebSocket, full-duplex channel. The JS compiler generates a `WebSocketX` class — the base every reactive action's generated code sits on top of — which extends the standard `WebSocket` with the exception that `message` and `send` are overridden to accept generic types, and the constructor takes a factory for message creation when working with JSON messages. Other generators (React, for instance) wrap the instance rather than replacing it, layering a `useWebSocketX` hook on top.

```
actions:
  - name: userStream
    url: ws://localhost:8081
    method: reactive
    in:
      fields:
        - {name: min, type: int}
        - {name: max, type: int}
        - {name: count, type: int}
    out:
      fields:
        - {name: number, type: int}
```

produces a complete typed request/response pair plus an action wrapper:

```
export class UserStreamAction {
  static URL = "ws://localhost:8081";
  static Method = "REACTIVE";
  static Create = (overrideUrl?, qs?, options) => {
    const Cls = options?.SocketClass ??
      WebSocketX<UserStreamActionReq, UserStreamActionRes>;
    return new Cls(url, undefined, {
      MessageFactoryClass: UserStreamActionRes,
    });
  };
}
```

The generated request (`UserStreamActionReq`) and response (`UserStreamActionRes`) classes are ordinary Emi DTO classes — same private fields, typed getters/setters, and auto-init guarantees as any other generated class (§8); the only thing `reactive` changes is the transport they travel over. The generated JavaScript works in both Node.js and browser environments, though the client side is the primary focus.

SSE gets the same treatment as WebSocket in spirit: a plain `get` *action whose response streams chunked messages compiles to typed messages on the client, and* `-tags react` *adds a matching* `useSse` *hook right alongside* `useWebSocketX`.

11. Swift & Kotlin Clients

Swift

The Swift compiler generates plain `Foundation`-based code (falling back to `FoundationNetworking` on Linux) — no third-party dependency pulled in. It produces type-safe request/response objects and an action wrapper per action, mirroring every other Emi compiler:

```
emi swift --path module.yaml --output ./Sources/Generated
```

`emi swift:dto` and `emi swift:headers` generate just the DTOs or just the headers, when the full compiler's output isn't needed. Each action carries its own metadata struct (name, URL, method) and a strongly typed response, so calling code never hardcodes a route or guesses a response shape:

```
struct GetSinglePostActionMeta {
    let name: String = "GetSinglePostAction"
    let url: String = "/posts/:id"
    let method: String = "Get"
}
struct GetSinglePostActionResponse {
    let statusCode: Int
    let headers: [String: String]
    let payload: Data?
}
```

Kotlin

The Kotlin compiler targets Android and JVM projects from the same definitions, using kotlinx.serialization for JSON and OkHttp for networking — both need to be on the consuming project's classpath, and nothing else does:

```
emi kotlin --path module.yaml \
  --output ./src/main/kotlin/generated
```

One invocation compiles everything a module declares — `dtos:`, `entities:` (each becomes a dto plus a set of CRUD actions), and `actions:` — into a flat directory of `.kt` files plus shared runtime files. `emi kotlin:dto` and `emi kotlin:headers` narrow the output the same way their Swift counterparts do. `-pkg` sets the Kotlin `package` every generated file is written under (default `unknownpackage`) — it only matters once more than one module compiles into the same project, so cross-module dto/entity references resolve to the right package.

Shared shape across both compilers

- `-path` / `-output` — same convention as every Emi sub-compiler
- Without `-output`, generated virtual files print as JSON to the terminal for inspection before anything touches disk
- `-tags` — comma-separated compile features, same mechanism as the Go and JS compilers

12. Additional Language Targets

Every remaining client target takes the same `.emi.yml` through its own sub-compiler, with the same `-path`/`-output` pair every other Emi compiler already uses. Each one below is client-only — no server, no CLI, no entity/gorm persistence — and most share the same two housekeeping tags: `no-sdk` (skip embedding the runtime files) and `no-pkg` (skip the language-native package manifest — `requirements.txt`, `pom.xml`, `.csproj`, `composer.json`, `pubspec.yaml`, `README.md`, depending on the target).

Python

`dataclasses`-based DTOs and an `httpx` client, synchronous by default:

```
emi python --path module.emi.yml --output ./out/python
emi python --path module.emi.yml --output ./out/python --tags
    async
```

`-tags async` swaps every generated client function to `async def`, backed by `httpx.AsyncClient`, instead.

Dart

DTO classes with hand-generated `toJson`/`fromJson` and a `package:http` client:

```
emi dart --path module.emi.yml --output ./out/dart
```

C#

`System.Text.Json`-based DTOs and an `HttpClient` transport:

```
emi csharp --path module.emi.yml --output ./out/csharp
```

Java

Jackson-based DTOs on top of `java.net.http.HttpClient`:

```
emi java --path module.emi.yml --output ./out/java
```

PHP

Reflection-based DTO hydration and a `curl` HTTP transport:

```
emi php --path module.emi.yml --output ./out/php
```

C

Vendored `cJSON` (de)serialization and `libcurl` transport — no nullable value types without pointers, a documented scope limit:

```
emi c --path module.emi.yml --output ./out/c
```

C++: generic and Unreal Engine

One compiler, two dialects, picked with `-dialect` (or the equivalent `-tags unreal`, handy when a caller is already building a `-tags` list): portable ISO C++17 for desktop/ESP-IDF/ Arduino by default, or `USTRUCT`/`UPROPERTY` DTOs usable from Blueprint when the Unreal dialect is selected.

```
emi cpp --path module.emi.yml --output ./out/cpp # generic ISO C
    ++17
emi cpp --path module.emi.yml --output ./out/cpp-unreal \
  --dialect unreal # Unreal Engine
```

`-path`, `-output`, and `-tags` is the interface every Emi compiler shares, this section's seven targets included — run `emi tags` at any time to print the full, current tag list for every registered target, straight from the compiler's own source of truth rather than a copy that can drift out of date here.

13. Exports: Markdown, OpenAPI, Postman

Three commands don't emit source code at all — they *describe* a module in a standard interoperable format, reading the same definition that produces the Go/Swift/Kotlin/JS clients, so none of them can drift away from the real implementation.

`emi md` — Markdown docs

The fastest way to get human-readable, always up to date documentation for a module: a title, description, table of contents, and one section per action listing method, URL, CLI name, and request/response fields.

```
emi md --path module.yaml --output ./out
# ./out/postsModule.md
```

Output for a single-action module looks like:

```
### `GetSinglePostAction`
| | |
|---|---|
| **Method** | `GET` |
| **URL** | `/posts/:id` |
| **CLI** | `get-single-post` |
#### Response
| Field | Type |
|---|---|
| `userId` | `int64` |
```

Committed next to the module, served in a docs site, or rendered straight into a README.

`emi openapi` — OpenAPI 3

A *describer*, not a compiler: reads the definition, produces a standard OpenAPI 3 document for Swagger UI, Redoc, code generators, or API gateways.

```
emi openapi --path module.yaml --output ./out --json
# ./out/postsModule.openapi.yaml (+ .json)
```

Route parameter syntax is rewritten (`/posts/:id` becomes `/posts/{id}`), path parameters are promoted, and Emi field types map onto the correct OpenAPI `type`/`format` pairs (e.g. `int64` → `type: integer, format: int64`).

`emi postman` — Postman v2.1 collection

Every action becomes a Postman request with method, URL, and the standard Emi headers pre-filled, ready to import and start hitting endpoints immediately.

```
emi postman --path module.yaml \
  --host localhost --port 8080 --output ./out
```

Host and port are emitted as Postman environment variables (`{{HOST}}`, `{{PORT}}`) so switching between local, staging and production is a matter of changing the active environment, and the common Emi headers (`authorization`, `role-id`, `workspace-id`) are wired to `{{AUTH}}`/`{{RID}}`/`{{WID}}` variables, filled once and reused across every request.

All three exporters share the same two flags: `-path` (the definition file) and `-output` (where to write; omit it and the result prints to stdout so it can be piped or inspected). Unlike the code-generating compilers (§12), none of the three currently registers its own `-tags` — there is nothing yet to toggle in a document that's just read back off the definition.

14. Vsql: Hand-Written SQL with Typed Bindings

A `vsql` entry pairs a hand-written SQL query with a typed parameter shape — the same governing idea as Query Predict (§15), reworked as a first-class part of the module instead of a separate command-line tool, and extended well past what Query Predict ever did: array/object/-complex parameters, an optional caller-controlled column projection, a typed JSON-Logic filter surface, and a debugging command. The philosophy carries over unchanged: **no schema introspection, no auto-migration, no SQL synthesis**. Emi never looks at a live database and never writes a query for you — it only makes the boundary between the SQL you wrote and the Go code that calls it type-safe. Entries live in a top-level `vsqls` list.

```
vsqls:
  - name: insertUser
    description: Inserts one user and returns the new row's id.
    params:
      - {name: email, type: string}
      - {name: role, type: enum, of: [{k: Admin}, {k: Member}]}
    columns:
      - {name: id, type: int64, selected: true}
```

```
    - {name: email, type: string, selected: false}
  query: |
    INSERT INTO users (email, user_role) VALUES ({{ sql .Email
        }}, {{ sql .Role }})
    RETURNING {{ .Columns.Cols }};
```

`emi go` compiles this into `InsertUserVsqlParams` (the Go struct generator dtos and actions already use, so nested objects, arrays, enums and `complex` fields all work identically here), a `InsertUserVsqlQuery` constant holding the SQL verbatim — template syntax and all — and `PrepareInsertUserVsql(params, columns)`, which just hands back `(query, args)` unrendered. Rendering the template and executing it against a real driver is left to the caller; `examples/vsql/vsql` ships a reference renderer (`sql`, `deref`, `sqlFields`/`sqlFieldsExcept` template helpers) worth copying wholesale into a real project rather than reinventing.

Params and In

`params` is the quick, inline way to declare input — most queries only need it. `in` is an action-shaped alternative, *alongside* `params` rather than instead of it: the exact type `EmiAction.in` uses, so it can reference an existing dto by name (`in: {dto: userFilterDto}`) or carry typed headers, instead of repeating fields inline. Preprocessing resolves `in` and merges it into `params` before any generator runs — `params` wins on a field-name collision, since it's the more specific, local declaration. Declare either one, or both; the merge is what makes a vsql's input shape reusable as an action's request body later, without a second, differently-shaped declaration.

Captures

`captures` pulls fields from an existing dto, template dto, or action body into `params` at preprocessing time — the same mechanism and `EmiCapture` shape used elsewhere in the module, with finer-grained `include`/`exclude` control than `in.dto`'s whole-dto reference. By the time a generator runs, captured fields have already been flattened into `params` and `captures` itself is cleared.

Columns: an optional, typed projection

`columns` declares which columns a query can optionally include — typically a `SELECT`/`RETURNING` projection, though nothing stops a query from repurposing the same `Selected` picker for a different kind of toggle (an `UPDATE` gating which `SET` clauses actually apply is a real pattern used in `examples/vsql-sqlite`). Each entry embeds `EmiField` directly, so a column has the full field vocabulary — `object`, `array`, `enum`, `complex`, nested `fields` — not just a name and a type.

```
columns:
  - name: preferences
    type: object
    selected: false
    column: "json_object('theme', theme, 'locale', locale)"
    fields:
      - {name: theme, type: string}
      - {name: locale, type: string}
```

EmiColumn fields

name / type / …	the full embedded `EmiField` — identifier and Go/emi type
column	SQL fragment to project when selected; defaults to `name` snake_cased. Any expression works, including one spanning *more than one* physical column (see below)
selected	default inclusion state

From a `columns` list, `emi go` generates four things: a `{Name}VsqlRow` response DTO (one field per column, run through the very same struct generator as `params` — an `object` column becomes a real nested struct, not a string); a `{Name}VsqlColumns` picker (one `emigo.ColumnState` per column) with a `New{Name}VsqlColumns()` constructor seeded from each `selected:` default and a `Cols()` method joining the currently-selected columns' `column` text into a projection, so a query template writes `{{ .Columns.Cols }}` once instead of repeating the column list; a `{Name}VsqlData` wrapper combining `Params` (embedded, so `{{ .Field }}` still resolves via Go's field promotion) with `Columns`, which is what a query template actually receives; and (§14.6) a query-string cast helper. Declaring `columns` at all changes `Prepare{Name}Vsql`'s signature to take a second `columns` argument — unconditionally, even for a query like the `UPDATE` case above whose text never references `.Columns.Cols` at all.

Nullability. A column that defaults to `selected: false` is forced onto its nullable type variant in the generated row DTO during preprocessing, *regardless of the type it was declared with* — `int64` becomes `emigo.Nullable[int64]`, `object` becomes `emigo.Nullable[RowAddress]`, and so on. A column a caller can choose to leave out has no guarantee of being populated once a result is scanned, so the field has to be able to represent "not fetched" on its own terms — distinguishing that from "fetched, zero value". `any` is left alone (already unconditionally nil-capable) and `complex` becomes `complex?`, a no-op for Go generation; a complex type has no wire-level nullability of its own; a `Money`-shaped type deciding what an absent amount looks like is its job, via the same `SQLValue() (string, bool)` contract the reference renderer's `SQLValuer` interface already uses.

Complex columns spanning several physical columns. Because `column` is a free-form SQL fragment, one `EmiColumn` (one `Selected` toggle) can legitimately project more than one real column:

```
  - name: money
    type: complex
    complex: Money
    selected: false
    column: "amount_cents, currency"
```

`Cols()` splices `"amount_cents, currency"` in verbatim under the one `Money` toggle. Emi's job stops at the field name and projection text; the actual multi-column `rows.Scan(&row.Money.AmountCents, &row.Money.Currency)` wiring is the consumer's, matching the "no schema introspection" stance — a two-column complex type is not something Emi could infer correctly on its own anyway.

Query text: inline, or read from a file

`query` compiles the SQL in as a Go string constant. As an alternative, `queryName` names a file to be read from a caller-supplied `fs.FS` *at runtime* instead — typically a real `.sql` file under `go:embed`, so the SQL gets real editor syntax highlighting and no YAML string-escaping. The two are mutually exclusive; declaring both is a compile-time error, not a silent "one wins". Setting `queryName` changes `Prepare{Name}Vsql` to take an `fs.FS` as its first argument and return an extra `error` (the file read can fail):

```
func PrepareSelectUsersPagedVsql(
    fsys fs.FS, params SelectUsersPagedVsqlParams, columns
        SelectUsersPagedVsqlColumns,
) (query string, args interface{}, err error)
```

Predict: detecting columns from a plain SELECT

Setting `predict: true` populates `columns` automatically at preprocessing time by parsing `query` (or the file named by `queryName`) as a plain SQL `SELECT` and detecting its projected columns — the exact column-detection mechanism Query Predict has always used, ported into a standalone `lib/sqlpredict` package (no dependency on the rest of Emi's core) so vsql can use it without Query Predict needing to stick around. It is a no-op whenever `columns` is already non-empty — a hand-written list always wins, never silently overwritten.

```
vsqls:
  - name: listActiveUsers
    predict: true
    query: |
      SELECT id, email, field(count(o.order_id), 'int64', '
          totalOrders', true)
      FROM users u LEFT JOIN orders o ON u.id = o.user_id
      WHERE u.id > {{ .MinId }}
      GROUP BY u.id;
```

`id`/`email` come out as plain `string` columns (the default when no type hint is given); the `field(expr, 'type', 'goName', optional)` marker — identical syntax to Query Predict's own — names `totalOrders` explicitly, types it `int64`, and its `true` fourth argument makes it optional, which comes out as `emigo.Nullable[int64]` in the generated row.

Predict: what it can and can't do

Only a plain SELECT	the underlying parser implements MySQL grammar and has no notion of `RETURNING` at all — an `INSERT/UPDATE/DELETE ... RETURNING` vsql, the shape most examples in this book actually use, needs `columns` written by hand instead
Template markup	every `{{ ... }}` block is replaced with `NULL` before parsing, so a `WHERE/VALUES` clause mixing in Go-template syntax still parses — only the `SELECT` list itself has to be real SQL
A wildcard select	`SELECT *` (bare, table-qualified like `u.*`, or mixed with named columns) leaves `columns` empty and returns *no error* — what `*` expands to depends on the live schema, which this project's SQL philosophy never inspects. The vsql simply ends up with no row DTO and no column picker, exactly as if `predict` had never been set
Joins with same-named columns	`u.id`/`o.id` keep their table qualifier in both the detected Go name (`UId`/`OId`) and the re-projected SQL (`u.id`, not a bare, ambiguous `id`) — an explicit `AS alias` is honored the same way, keeping the full `expr AS alias` text rather than collapsing to the bare alias
Genuine collisions	if two columns still resolve to the same Go name after that, preprocessing fails with a clear error rather than generating a struct with a duplicate field

Casting the column picker from a query string

Whenever `columns` is declared, `emi go` also generates `{Name}VsqlColumnsQuery` — reusing the exact same `emigo.UnmarshalQs`/`emigo.NewDecoder` machinery an action's own `qs` fields already use — so a caller can build the picker straight from an incoming HTTP request instead of translating query parameters into it by hand:

```
cols := SelectUserByIdVsqlColumnsQueryFromHttp(r).Columns()
query, args := PrepareSelectUserByIdVsql(params, cols)
```

`Columns()` only overrides the keys the query string actually mentioned (checked via `url.Values.Has`); anything absent keeps its own declared `selected:` default rather than becoming `false` — a client asking for `?email=true` gets the usual defaults plus email, not just email alone.

Filters: typed JSON-Logic queries

`filters` declares which fields a caller may reference inside a JSON-Logic filter expression (https://jsonlogic.com) run against this query's rows, e.g. `{"==": [{"var": "role"}, "Admin"]}` — the same JSON-Logic-to-SQL path this ecosystem already has for entity Browse actions (`emigorm.ApplyQueryFilter`, via `jsonlogic2sql`), now with an allow-list Emi can generate rather than an unconstrained string. `filters` is deliberately a plain `[]*EmiField`, not `[]*EmiColumn`: an allow-list is a value shape (name + type), not a selection picker, so it compiles through the very same struct/class generator as `params` —

GoCommonStructGenerator in Go, JsCommonObjectGenerator in JS — rather than reusing Columns' picker machinery.

When left empty and columns is set, filters defaults to columns' own fields during preprocessing — the row surface a SELECT already exposes is the natural default set of things worth filtering on. A query with no columns needs filters declared explicitly if it wants filtering at all.

```
vsqls:
  - name: selectUserById
    columns:
      - {name: id, type: int64, selected: true}
      - {name: email, type: string, selected: true}
      - {name: role, type: enum, of: [{k: Admin}, {k: Member}],
          selected: false}
    query: |
      SELECT {{ .Columns.Cols }} FROM users WHERE id = {{ .Id
          }};
```

emi go generates SelectUserByIdVsqlFilters (defaulted from the three columns above), plus a runtime allow-list — Go generics can't express a string-literal union the way TypeScript can, so this is the compile-time check Go gets instead:

```
var SelectUserByIdVsqlFilterFields = []string{"id", "email", "
    role"}

func SelectUserByIdVsqlFilterFieldAllowed(name string) bool {
  for _, f := range SelectUserByIdVsqlFilterFields {
    if f == name {
      return true
    }
  }
  return false
}
```

A real caller runs this before ever handing a filter to jsonlogic2sql: walk the parsed JSON-Logic tree, and reject the request if any {"var": ...} leaf names something outside SelectUserByIdVsqlFilterFields.

TypeScript gets more: a compile-time check. Under -tags typescript, the same field list also becomes a literal string union, plus one minimal JSON-Logic node type constrained to it:

```
export type SelectUserByIdVsqlFilterField = "id" | "email" | "
    role";

export type SelectUserByIdVsqlFilterVarNode = {
  var: SelectUserByIdVsqlFilterField;
};
export type SelectUserByIdVsqlFilterOperatorNode = {
  [operator: string]: SelectUserByIdVsqlFilterValue;
} & { var?: never };
export type SelectUserByIdVsqlFilterValue =
  | SelectUserByIdVsqlFilter
  | string | number | boolean | null
  | SelectUserByIdVsqlFilterValue[];
export type SelectUserByIdVsqlFilter =
  | SelectUserByIdVsqlFilterVarNode
  | SelectUserByIdVsqlFilterOperatorNode;
```

A caller can now write real JSON-Logic and have the compiler catch a typo or a field the query never declared:

```
const ok: SelectUserByIdVsqlFilter = {
  and: [{ "==": [{ var: "role" }, "Admin"] }, { "!": { var: "
      email" } }],
};

// @ts-expect-error - "isAdmin" was never declared on this query
const bad: SelectUserByIdVsqlFilter = { var: "isAdmin" };
```

Three things that look obvious and aren't

No third-party JsonLogic types	JsonLogic's own spec ships no TypeScript typing, and a generic library couldn't know a *specific* query's allowed fields anyway — only Emi, generating per-vsql, can. The generated type stays plain-JSON-shaped, though: nothing stops handing a value of it straight to a real runtime such as json-logic-js
A bare index signature isn't enough	{ [operator: string]: ... } alone lets { var: "notAField" } slip through as just another operator key — TypeScript's structural typing doesn't single out "var". & { var?: never } on the operator-node arm is what actually forces a var-carrying object to be checked against the field-name union instead; verified against a real tsc run, not just read off the type
Needs strictNullChecks	without it, a literal like "email" widens to plain string inside the recursive union and the constraint silently stops rejecting anything — also caught by an actual tsc run, not assumed

examples/vsql-columns/ts has a working tsconfig.json and a filter-typesafety.ts exercising exactly this: valid JSON-Logic typechecking clean, three @ts-expect-error cases (an unknown field nested, one as a direct var node, and a typo of a real field name) that must keep failing or the whole file fails to typecheck — make test-ts regenerates and runs it for real.

Debugging a query: emi vsql:debug

Renders one vsql's final SQL against caller-supplied values and prints it, with no compiled project, no generated Go code, and no database — useful while writing a definition, to check the template even parses and the projection comes out right.

```
emi vsql:debug --path queries.emi.yml --name insertUsers \
  --params '{"users":[{"email":"a@x.com"}]}' --select "id,email,
      money"
```

-params and -in are each a JSON object of field values (merged, -params winning a collision, mirroring params/in's own precedence); -select is a comma list of column names to mark selected, replacing every column's own default entirely when given. The command builds the same template-data shape (.Field, .Array.Items, .Columns.Cols, .Columns.X.Selected) real generated Go code would, directly off the parsed schema plus the given JSON — no compiled struct in the loop at all.

A recurring template gotcha

Go's `text/template` truthiness rules only treat `0`, empty strings, `nil`, and zero-length arrays/slices/maps as false — a `struct` is never false, regardless of its fields. That means `{{ if .Email }}` on an `emigo.Nullable[string]` param is *always* true, whether or not it was actually set. Guard a nullable field's presence with `{{ if .Email.IsSet }}` instead; `{{ sql .Email }}` on its own already renders correctly either way (the reference renderer's `sql` helper duck-types `Nullable[T]` and omits/nulls it appropriately), but a whole conditional clause built around one needs the explicit `.IsSet` check.

Every artifact vsql generates — params, captures, columns, the row DTO — goes through the exact same field/struct/complex-type machinery as a dto or an action body. A vsql query is never a second type system bolted onto the first one; it is hand-written SQL wearing the same typed clothes as everything else Emi compiles.

`examples/vsql`, `examples/vsql-columns` (Go and, under `ts/`, a real generated TypeScript SDK) and `examples/vsql-sqlite` show the whole surface end to end, the last one running ten queries — batch inserts, optional filters, partial updates, a predicted join — for real against an in-memory SQLite database.

15. Query Predict

Query Predict (`qp`) is Emi's approach to **hand-written SQL with generated, type-safe Go bindings**. You write plain `.sql` files; Emi parses them, predicts the shape of each result row and the parameters it needs, and generates a typed row struct and a prepared-statement builder. There is no schema introspection, no auto-migration, and no SQL synthesis — the SQL you write is the SQL that runs. This keeps the full power of SQL in your hands while still giving compile-time safety over the columns and parameters a query touches.

Three commands

emi qp:dir — scans a directory of `.sql` files, one generated Go file per query, named from the SQL file.
emi qp:gen — reads a Query Predict YAML document bundling several named queries and a Go package name.
emi qp:sql — compiles one query into executable Go and transforms its metadata into pure SQL.

```
emi qp:dir --path ./queries-source --output ./output
```

For a query like `select name, id from users;`, Emi emits a typed row struct, the original SQL as a constant, and a context/prepare helper:

```
const BasicSelectSQL = `select name, id from users;`
type BasicSelectRow struct { Name string; Id string }
type BasicSelectContext struct {
  Filter, Having, Restriction string
  Params map[string]interface{}
}
```

SQL helper functions

A small set of marker functions can appear inside the SQL itself, letting dynamic behaviour stay in the query while the generated code remains predictable and typed:

Query Predict SQL helpers

`field(expr, 'type', 'goName'?)`	annotate an expression's Go type (and optionally its field name) when it can't be inferred
`useval('name')`	a named value substituted from `ctx.Params` at runtime, safely escaped
`filter()`	replaced by the `Filter` clause from the context (default `1`)
`restriction()`	replaced by the `Restriction` clause (default `1`)

```
SELECT u.user_id,
  field(u.user_name, 'string') as UserName,
  field(COUNT(o.order_id), 'int64') AS total_orders
FROM users u LEFT JOIN orders o ON u.user_id = o.user_id
WHERE filter()
GROUP BY u.user_id, u.user_name
LIMIT useval('limit')
```

Here `field(..., 'int64')` tells the generator the aggregate column is an `int64`, `filter()` is swapped for whatever filter the caller supplies at runtime, and `useval('limit')` pulls the `limit` entry out of `ctx.Params` — Emi never invents schema, migrations, or queries of its own; it only makes the boundary between hand-written SQL and Go type-safe.

16. Running the Compiler in the Browser

Because the Go server compiles to WebAssembly and Postgres (pglite) does too, Emi can run an **entire backend inside a browser tab** — the *same* Go handlers, the *same* SQL, the *same* generated DTO classes — talking to a real Postgres database, with no server process and no network. It is not a separate, browser-only rewrite: compile the Emi-generated Go server to `GOOS=js GOARCH=wasm`, give it an HTTP router that lives in memory, and back its database calls with a real Postgres compiled to WebAssembly. The browser then "fetches" from a server sitting in the same tab.

In one sentence

`main.js` calls `localFetch()` into Go WASM's `net/http` router, which runs the ordinary generated handlers, which query a `Database` interface backed by `database-bridge.js` talking to pglite — everything from the handler down to the SQL string is byte-for-byte identical to what runs on the real Gin server in `cmd/server`.

What is shared and what is swapped

The whole design rests on one small interface — handlers never import `pgx` directly or hold a concrete connection, only a `Database`:

```
type Database interface {
  QueryRow(ctx context.Context, sql string, args ...any) pgx.Row
  Query(ctx context.Context, sql string, args ...any) (pgx.Rows,
       error)
  Exec(ctx context.Context, sql string, args ...any) (pgconn.
       CommandTag, error)
  CopyFrom(ctx context.Context, tableName pgx.Identifier,
    columnNames []string, rowSrc pgx.CopyFromSource) (int64,
         error)
}
```

Because this is exactly the shape of `*pgx.Conn`, the **server build** just passes a live connection straight through (passthrough). The **browser build** passes a `WasmDatabase` that routes the same calls through a JS bridge into pglite instead — the handler code above this interface never knows, or needs to know, which one it's talking to.

The wasm router

There is no wasm-specific router — the wasm build registers the exact same generated handlers on a plain `net/http.ServeMux` the real Gin server would use elsewhere; only what *drives* that mux differs. `emigo.LiftWasmServer(mux, config)` is the entire bridge: it exposes one global JS function (`window.handleWasmRequest` by default) that turns a fetch-style call into a real `*http.Request` and captures the response with an in-memory `httptest.ResponseRecorder`:

```
js.Global().Set(jsFunctionName, js.FuncOf(func(_ js.Value, args
     []js.Value) any {
  method, url, bodyStr, headersJSON := args[0].String(), args
       [1].String(),
    args[2].String(), args[3].String()

  return js.Global().Get("Promise").New(js.FuncOf(func(_ js.
       Value, p []js.Value) any {
    resolve, reject := p[0], p[1]
    go func() {
      req := httptest.NewRequest(method, url, strings.NewReader(
           bodyStr))
      // ...headersJSON unmarshalled onto req.Header...
      rec := httptest.NewRecorder()
      mux.ServeHTTP(rec, req) // real routing, real handler,
           real ResponseWriter
      // ...rec.Result() marshalled to {status, headers, body}
           JSON, resolve.Invoke(...)...
    }()
    return nil
  }))
```

```
}))
```

Two details matter more than they look:

- **It returns a Promise, not the response.** A handler may call back into JS itself (a pglite query is a JS promise under the hood), and those promises only settle while the JS event loop is running. A synchronous `js.FuncOf` callback never yields that loop, so `ServeHTTP` runs on a goroutine instead — control returns to JS immediately, the event loop keeps turning, and the goroutine's own awaited promises can resolve.
- **The router never sees JS at all.** `httptest` builds a genuine `*http.Request`/`ResponseRecorder` pair, so `mux.ServeHTTP` runs precisely the code path a real inbound request would — same routing, same middleware, same generated handler.

On the JS side, the bridge is a fetch-shaped wrapper around that one function:

```
async function localFetch(url, opts = {}) {
  const raw = await window.handleWasmRequest(
    opts.method || "GET", url, opts.body || "",
    JSON.stringify(opts.headers || {}),
  );
  const { status, headers, body } = JSON.parse(raw);
  const h = new Headers();
  for (const [k, vs] of Object.entries(headers || {})) for (
       const v of vs) h.append(k, v);
  return new Response(body, { status, headers: h });
}
```

Because it hands back a real `Response`, this slots straight into `FetchxContext`'s `fetchOverrideFn` (§8.6): point the generated SDK's transport at `localFetch` instead of the browser's native `fetch`, and every generated hook, DTO, and action wrapper — unmodified — talks to the in-tab Go server instead of the network, with no code aware of the difference.

`reactive` actions don't go through this router at all: a hijackable socket doesn't exist in wasm, so `emigo.NewWasmReactor()` bridges the same generated reactive handler straight to a `WebSocketWasm` JS class instead — a parallel path, not a detail of the HTTP one above.

> *See it running: the in-browser server demo — arguably the clearest possible demonstration of what "one definition, real code everywhere" actually buys a team: the exact same handlers ship to production and run, unmodified, in a visitor's browser tab.*