Go

Next.js

Grit: The Full-Stack

Go and Next.js, batteries and optimized for AI.

GRIT

Meta-Framework

optimized for AI.

Turborepo

The Grit Philosophy

Full-Stack Fusion

A Go backend and Next.js frontend unified by shared TypeScript types.

Batteries Included

Everything from authentication to background jobs and deployment configurations is pre-configured.

AI-Optimized

Deterministic code generation with strict conventions designed as a canonical reference for LLM agents.

One CLI command scaffolds a complete production-ready project.

The Technology Matrix

Backend

Go 1.24, Gin, GORM, PostgreSQL, Redis.

Frontend

Next.js 15, React 19, TypeScript, Tailwind CSS.

Admin

Custom Filament-like React panel.

Shared Layer

Zod schemas + TypeScript types.

Infrastructure

Docker Compose, Turborepo, pnpm.

Observability & Security

Pulse (Tracing) & Sentinel (WAF).

Turborepo Monorepo Architecture

Thin handlers, fat services. Models and DB logic isolated here.

App router, optimized for public-facing ISR.

Protected routes, generated React Query hooks.

The bridge. Generated Zod schemas and TS types keep frontend and backend perfectly synced.

The CLI Engine

```
$ grit new <app-name>
```

Scaffolds the full monorepo (or use --api for backend-only).

```
$ grit start server
$ grit start client
```

Starts the Go API and the Turborepo frontends in separate terminals (Note: there is no grit dev).

```
$ grit sync
```

Regenerates shared Zod schemas and TypeScript types after manual Go model edits.

Every command is idempotent—safe to re-run at any time.

Full-Stack Code Generation

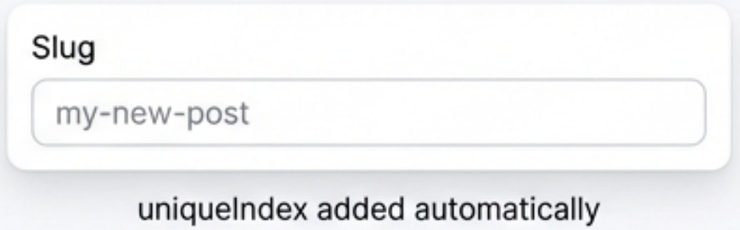
```
grit generate resource Product
--fields "name:string, price:float64"
```

Grit Engine

- .go — Go Models & DB Migrations
- .go — API Routes & Handlers
- .go — Business Logic Services
- .ts — Shared Zod Schemas
- .ts — React Query Hooks
- .tsx — Admin UI Pages

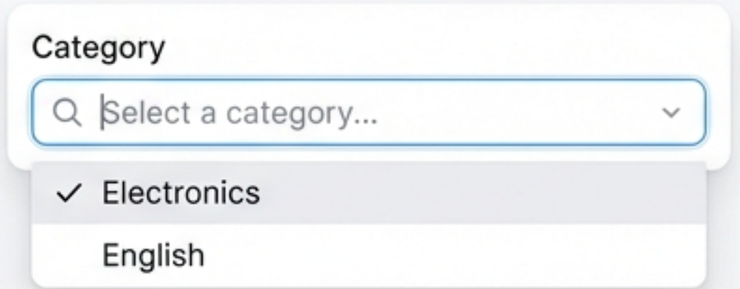
Relationships & Auto-Slugs

```
slug:slug
```

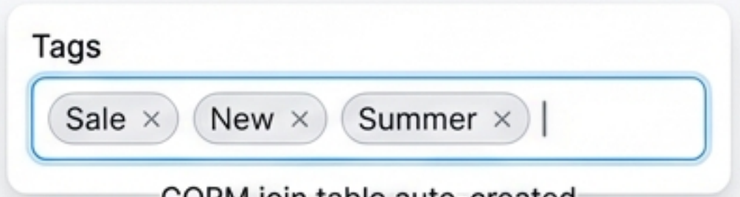


uniqueIndex added automatically

```
category_id:uint:fk:Category
```




```
tag_ids:[]uint:m2m:Tag
```

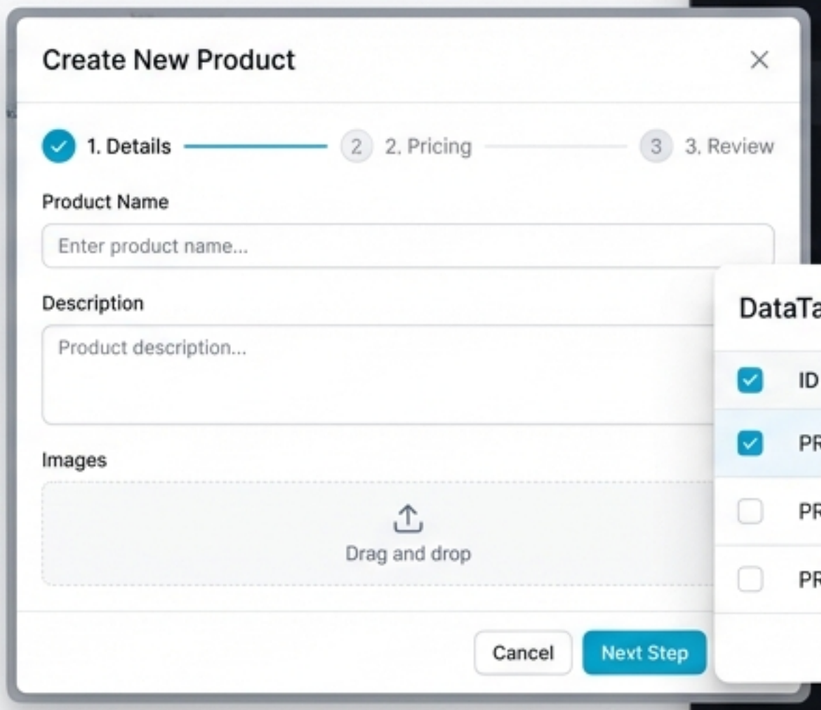


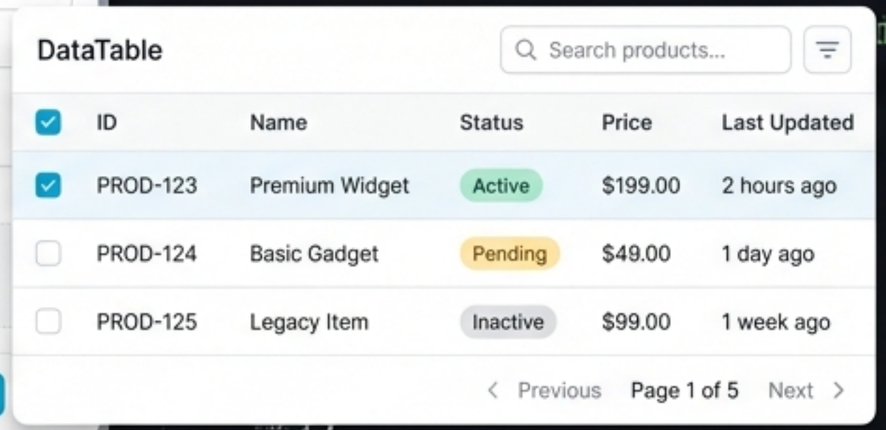
GORM join table auto-created

Admin Panel & UI Primitives

```
import { defineResource } from 'your-admin-framework';

defineResource({
    name: 'Product',
    fields: { ... },
    formView: 'modal-steps',
});
```





Standalone ready: FormBuilder, DataTable, and FormStepper components can be used on any page, independent of the resource system.

Built-In Backend Batteries

Storage
Presigned URL uploads to S3/R2/MinIO.

Email
Resend integration with local Mailhog dev server.

Jobs
Redis-backed async queues (Asynq) for heavy lifting.

Cache
Redis caching middleware for instantaneous API responses.

Sentinel
Built-in WAF, rate limiting, and brute-force protection.

Pulse
Tracing, DB monitoring, and Prometheus observability.

Tuned for Scale: Backend Performance

Gzip Compression

Automatically compresses JSON payloads by 60–80%.

Tuned DB Pool

Prevents connection exhaustion (`MaxIdle: 10`, `MaxOpen: 100`).

Cache-Control

Public endpoints emit headers for CDN/edge caching.

Direct Uploads

Binary uploads go directly to storage (S3/R2) via presigned URLs, bypassing the Go API memory completely.

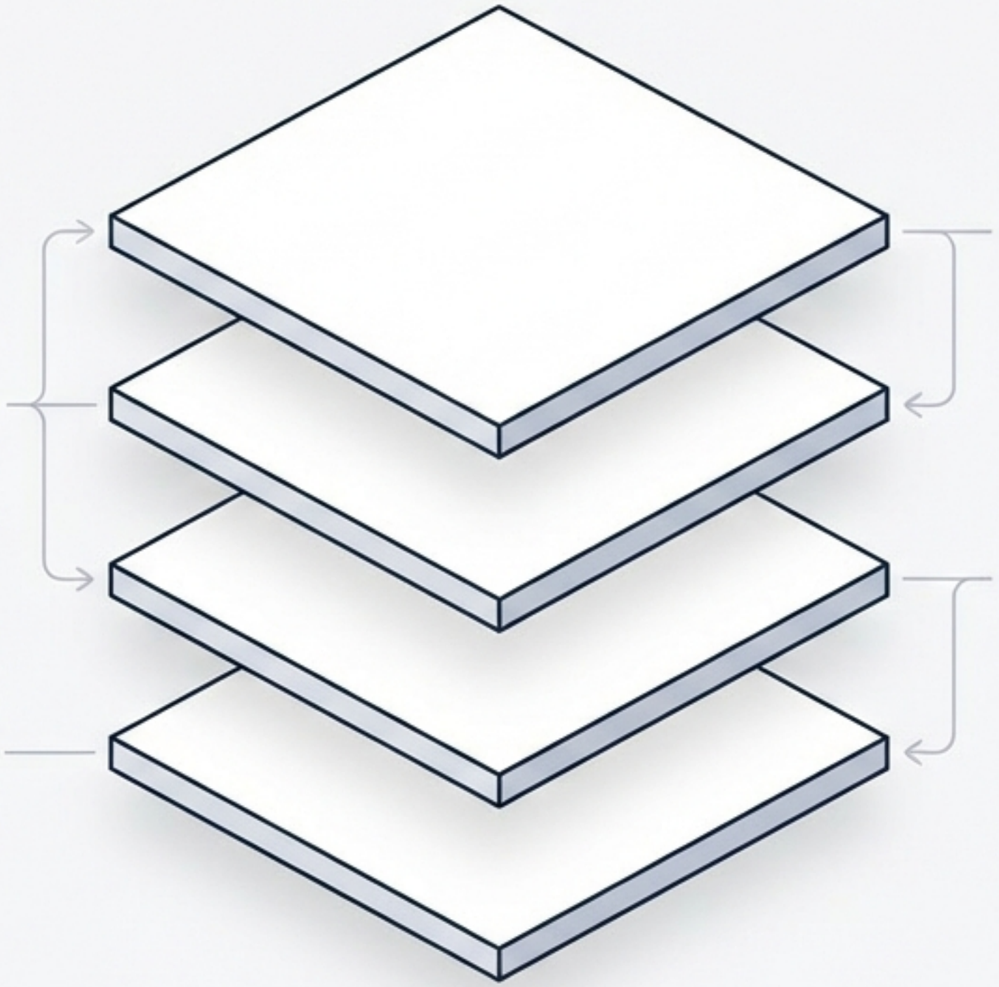
Blazing Fast User Experiences

Edge

Incremental Static Regeneration (ISR) serves cached HTML in milliseconds for public content.

Build

Turborepo Build Cache cuts CI times from minutes to seconds.



Client

React Query handles smart client-side caching and instantaneous list refreshes in the Admin panel.

Server

React Server Components (RSC) fetch data with zero JS bundle cost.

Deployment Flexibility: Docker vs. Cloud Native

The Default Setup (Docker)

docker-compose.yml

```yaml
version: '3.8'
services:
  db:
    image: postgres:latest
  redis:
    image: redis:alpine
  minio:
    image: minio/minio
  mailhog:
    image: mailhog/mailhog
```

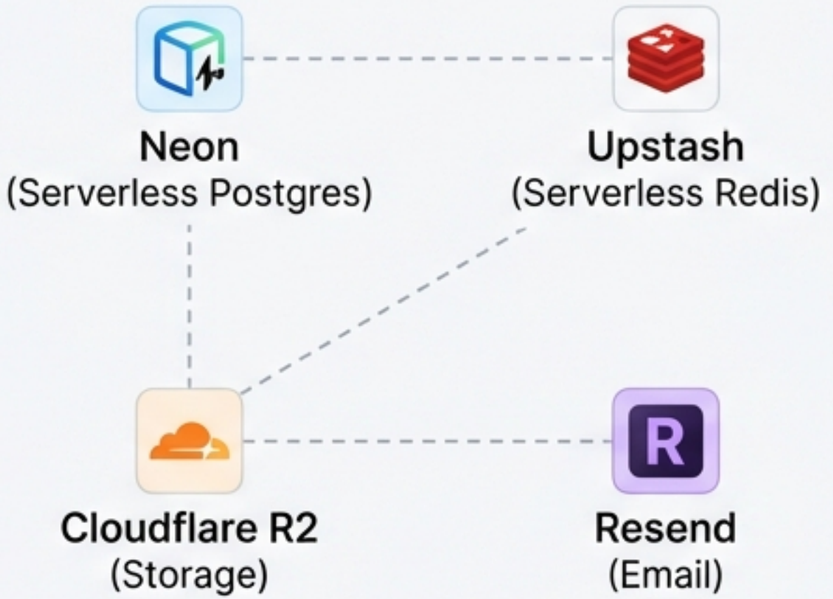
PostgreSQL

Redis

MinIO & Mailhog

Best for powerful local machines.

The No-Docker Setup (Cloud Native)



Best for low-spec machines and seamless prod deployment.

The API Standard

```
{
  "data": {...},
  "message": "Product created successfully"
}
```

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "name is required"
  }
}
```

Predictability is non-negotiable. Frontend hooks and admin components depend on this exact shape. Status codes (422, 404, 401, 500) map strictly to error types.

The Golden Rules

1 Never remove GRIT: markers

`// GRIT:MODELS` and `// GRIT:ROUTES` are permanent CLI injection points.

2 Never use multipart/form-data

Always use presigned URLs for uploads.

3 Keep handlers thin

All GORM queries belong in services, never in handlers.

4 Sync shared types

Always run `grit sync` after manually editing Go models to prevent drift.

Start Building

```
$ go install github.com/MUKE-coder/grit/cmd/grit@latest

$ grit new myapp
```

Read the Docs View on GitHub