Lesson 2: Go Project Structure

Setup, Modules, Directory Conventions & Essential Commands

The Mindset Shift from TypeScript / Node.js

Coming from TypeScript, you're used to **npm init**, a **package.json**, and a folder structure like `src/api/routes/services/` with Prisma in the root. Go works very differently. There's no central package registry like npm, no `node_modules`, and the compiler itself enforces certain structural rules (like the `internal/` directory). Understanding these differences early will save you a lot of frustration.

Quick Comparison: TS/Node vs Go Setup

Concept	TypeScript / Node.js	Go
Init a project	`npm init`	`go mod init github.com/user/proj`
Manifest file	`package.json`	`go.mod`
Lock file	`package-lock.json`	`go.sum`
Install deps	`npm install express`	`go get github.com/gin-gonic/gin`
Clean deps	`npm prune`	`go mod tidy`
Dependency dir	`node_modules/ (local)`	`~/go/pkg/mod/ (global cache)`
Run code	`npx ts-node src/index.ts`	`go run cmd/server/main.go`
Build binary	`tsc + bundler`	`go build -o myapp cmd/server/main.go`
Test	`jest / vitest`	`go test ./...`
Formatting	`prettier (config needed)`	`go fmt ./... (built-in, zero config)`
Linting	`eslint (config needed)`	`go vet ./... (built-in)`

Key difference: Go does not copy dependencies into your project folder. All downloaded modules live in a global cache at `$GOPATH/pkg/mod`. Your `go.sum` file is a cryptographic checksum of every dependency — similar to a lock file but focused on integrity verification.

Setting Up a Go Project (Step by Step)

Step 1: Create the directory and initialize the module

```
mkdir myapi && cd myapi
go mod init github.com/yourname/myapi
```

This creates a **go.mod** file — Go's equivalent of package.json. The module path should be the repository URL where the code will live. Even for local projects, use a meaningful path.

Step 2: What go.mod looks like

```
module github.com/yourname/myapi

go 1.22
```

```
require (
    github.com/gin-gonic/gin v1.9.1
    github.com/jmoiron/sqlx v1.3.5
)
```

Step 3: Add a dependency

```
go get github.com/gin-gonic/gin
go get github.com/lib/pq@v1.10.9   # specific version
```

Unlike npm, you don't need a separate install step. When you `import` a package in your code and run `go mod tidy`, Go automatically resolves and downloads it.

Step 4: Build and run

```
go run cmd/server/main.go    # run without building a binary
go build -o myapi cmd/server/main.go  # compile to binary
./myapi                     # run the binary directly
```

Go Project Structures (By Scale)

Go has **no single mandated structure**. The official Go team explicitly says there is no "standard layout". However, strong conventions have emerged. The right structure depends on your project's size.

1. Flat Structure — Small Projects / CLI Tools / Scripts

```
myapp/
|-- go.mod
|-- go.sum
|-- main.go
|-- handler.go
|-- db.go
`-- README.md
```

All files declare `package main`. No subdirectories. This is perfectly valid and is how many successful open-source tools start. Don't add structure you don't need yet.

2. Standard Layout — Medium to Large Services

```
myapi/
|-- cmd/
|   `-- server/
|       `-- main.go           # entry point
|-- internal/
|   |-- handler/             # HTTP handlers (like controllers)
|   |   |-- user.go
|   |   `-- order.go
|   |-- service/             # business logic
|   |   |-- user.go
|   |   `-- order.go
|   |-- repository/          # database layer (like Prisma)
|   |   |-- user.go
|   |   `-- order.go
|   |-- model/               # data structs (like TS interfaces)
|   |   `-- user.go
|   |-- middleware/
|   |   `-- auth.go
|   `-- config/
|       `-- config.go
|-- pkg/                     # public reusable packages (optional)
|   `-- logger/
|-- api/                     # OpenAPI specs, proto files
|-- configs/                 # YAML/TOML config files
|-- migrations/              # SQL migration files
|-- scripts/
|-- Makefile
|-- Dockerfile
|-- go.mod
|-- go.sum
`-- README.md
```

3. Multi-Binary / Monorepo Structure

When your repo produces multiple applications (an API server, a worker, a CLI tool), each gets its own entry under cmd/:

```
cmd/server/main.go    # the API server
cmd/worker/main.go    # background job processor
cmd/migrate/main.go   # database migration CLI
```

```
cmd/cli/main.go       # admin CLI tool
```

All binaries share the same `internal/` and `pkg/` code. This is one of Go's strengths — a single repo can produce multiple deployment artifacts trivially.

Directory Conventions Explained

cmd/ — Application Entry Points

Each subdirectory under `cmd/` is a separate binary. The `main.go` file inside should be thin — it wires up dependencies and starts the server. Think of it as your `index.ts` that imports everything and calls `app.listen()`.

internal/ — Private Application Code (Compiler-Enforced!)

This is Go's **killer feature** for project structure. Code inside `internal/` **cannot be imported by any other module**. This is enforced by the Go compiler itself — not a convention, not a linter rule, but a hard compiler error. There's no equivalent in TypeScript. This is where your handlers, services, repositories, models, middleware, and config logic live.

> **Why this matters:** You can freely refactor anything inside `internal/` without worrying about breaking external consumers. It's your safe zone for iteration.

pkg/ — Public Reusable Packages (Optional, Debated)

Code in `pkg/` signals "this is meant to be imported by other projects." Widely used (Kubernetes, Docker) but debated. If you're building a closed API service that nobody imports, you likely don't need it. Use `internal/` instead.

api/ — API Definitions

Contains OpenAPI/Swagger YAML files, Protocol Buffer `.proto` files, or JSON schema definitions. Not Go code — just specs.

configs/, migrations/, scripts/

configs/ holds environment-specific YAML/TOML/JSON config files (like `.env` files but structured). **migrations/** holds SQL migration files (similar to Prisma's migration folder). **scripts/** holds build, deploy, and maintenance shell scripts.

Makefile — Go's "npm scripts"

Go doesn't have a scripts section in go.mod. Instead, the community universally uses **Makefiles**:

```
.PHONY: build run test lint migrate

build:
    go build -o bin/server cmd/server/main.go

run:
    go run cmd/server/main.go

test:
    go test ./... -v

lint:
    golangci-lint run

migrate:
    go run cmd/migrate/main.go
```

Then you just run `make build`, `make test`, etc.

Mapping Your TS Mental Model to Go

Your TS Concept	Go Equivalent	Notes
`src/routes/`	`internal/handler/`	Handlers receive HTTP requests, call services
`src/services/`	`internal/service/`	Business logic layer, same concept
`src/middleware/`	`internal/middleware/`	Auth, logging, CORS — identical concept
`Prisma schema`	`internal/model/ structs`	Go structs with db tags replace Prisma models
`Prisma queries`	`internal/repository/`	Write SQL directly or use sqlx/GORM
`src/types/`	`internal/model/`	Go structs serve as both types and models
`.env`	`configs/ + os.Getenv()`	Or use viper/envconfig libraries
`package.json scripts`	`Makefile`	make build, make test, make run
`tsconfig.json`	`N/A`	Go has no config for compilation
`node_modules/`	`~/go/pkg/mod/`	Global cache, not per-project
`index.ts (entry)`	`cmd/server/main.go`	Thin entry point, wires deps
`Express / Fastify`	`Gin / Echo / Chi`	Popular HTTP router frameworks

Essential Go Commands Reference

Command	What It Does
`go mod init <path>`	Create a new module (like npm init)
`go get <pkg>`	Add or update a dependency
`go get <pkg>@v1.2.3`	Pin a specific version
`go mod tidy`	Remove unused deps, add missing ones (run often!)
`go mod download`	Download all deps to local cache
`go mod vendor`	Copy deps into vendor/ for reproducible builds
`go mod graph`	Show the full dependency tree
`go mod verify`	Verify checksums of downloaded deps
`go build ./...`	Compile all packages (catches errors)
`go build -o bin/app cmd/server/main.go`	Build a named binary
`go run cmd/server/main.go`	Compile and run (for development)
`go test ./...`	Run all tests in all packages
`go test -v -run TestName ./pkg/...`	Run specific test with verbose output
`go test -cover ./...`	Run tests with coverage report
`go test -bench . ./...`	Run benchmarks
`go fmt ./...`	Format all code (canonical style)
`go vet ./...`	Static analysis for common bugs
`go doc fmt.Println`	View documentation for any symbol

`go generate ./...`	Run code generation directives
`go work init`	Create a workspace (multi-module monorepo)
`go env GOPATH`	Show Go environment variables

How Testing Works in Go

In TypeScript you install Jest or Vitest, configure them, and put tests in a `__tests__` folder. In Go, testing is **built into the language and toolchain**. There's nothing to install.

Convention:

- Tests live **next to the code they test**, in the same directory:

```
internal/service/user.go       # your code
internal/service/user_test.go  # your tests
```

- Test files must end in `_test.go`
- Test functions must start with `Test` and take `*testing.T`:

```
func TestCreateUser(t *testing.T) {
    result := CreateUser("Alice")
    if result.Name != "Alice" {
        t.Errorf("expected Alice, got %s", result.Name)
    }
}
```

No assertions library needed (though `testify` is popular). Table-driven tests are the idiomatic pattern for testing multiple cases.

How Go Packages Work (vs TS Modules)

In TypeScript, every file can export and import independently. In Go, **a directory IS a package**. All `.go` files in the same directory must declare the same package name. There is no `export` keyword — instead, **capitalization controls visibility**:

Go Name	Visibility	TS Equivalent
`CreateUser`	Exported (public)	`export function createUser`
`createUser`	Unexported (private)	`function createUser (no export)`
`User`	Exported struct	`export interface User`
`user`	Unexported struct	`interface User (no export)`

Circular imports are forbidden. If package A imports package B, then B cannot import A. The compiler will reject it. This forces clean dependency graphs and is one of the reasons Go compiles so fast. In TS, circular imports silently cause subtle bugs; Go eliminates them entirely.

Important Rules and Gotchas

- **One package per directory** — Every .go file in a directory must declare the same package. No mixing.
- **Package name = directory name** — By convention (not enforced), the package name matches its directory.
- **No circular imports** — Compiler-enforced. Design your dependency graph carefully.
- **internal/ is compiler-enforced** — Not just convention. External modules literally cannot import it.
- **Tests live beside code** — Not in a separate tests/ directory. The _test.go suffix is special.
- **Unused imports are compiler errors** — You can't leave unused imports. Use _ for side-effect imports.
- **go.sum should be committed** — It's your integrity guarantee. Never delete it to "clean up".
- **Start flat, add structure as needed** — Don't create cmd/internal/pkg for a 200-line tool.

Next Lesson: Now that you understand how Go projects are structured, Lesson 3 will cover Go's type system, structs, interfaces, and error handling — the core language features you'll use every day.