Clean Architecture in Go

BRIDGING PRAGMATISM AND PRINCIPLES

Ever opened a ~~data file~~ function so long that your code editor started lagging?

//Sorry - The apology from the past

C# ObjectProcessor.cs

```
// Sorry.
public class ObjectProcessor
    {
        public static void ProcessData(
            DataTable dt,
```

Once upon a time in 2015

Congrats!

This legacy project is all yours!

```
|— Controllers/
|    ├— All.cs
|    ├— AuthController_new.cs
|    ├— HomeController.cs
|    ├— HomeCtr.cs
|    ├— UserController_old.cs
|— Core/
|    ├— BusinessLogic.cs
|    ├— BusinessLogic_V2.cs
|    ├— HelperFunctions.cs
|    ├— HelperFunctions_Backup.
|    |    |— Misc/
|    |    |    ├— MiscHelper.cs
|    |    |    ├— MiscStuff.cs
|— Repositories/
|    ├— AuthUser.cs
|    ├— UserRepo.cs
|    ├— UserRepo_new.cs
|    ├— OtherUser.cs
```

```
|— Shared/
|    ├— SharedUtils.cs
|    ├— Common.cs
|    ├— CommonHelperFunctions.cs
|— Common/
|    ├— CommonService.cs
|    ├— CommonDataHelper.cs
|    ├— CommonLogic.cs
|— Models/
|    ├— UserModel.cs
|    ├— UserModel_old.cs
|    ├— DataModel_final.cs
|— Data/
|    ├— DbHelper.cs
|    ├— DatabaseAccess.cs
|    ├— DatabaseHandler.cs
|— Middleware/
|    ├— Middleware.cs
|    ├— MiddlewareHandler.cs
|— Utils/
|    ├— Logger.cs
|    ├— UtilsHelper.cs
|    ├— UtilsFunctions_old.cs
```

The Fix It or Burn With It Situation

The project grew.
The team grew.
And so did the risk.

Clean Architecture - The Game Changer

The Classic Overengineering

"The number of layers is your choice"

"Abstraction is your greatest tool"

Giving second chances

Iterate
Simplify
Repeat

.. and then we got it
~~perfect~~ right

Beyond the first success

🤩 2017: First project, major refactoring success

🌟 2018: Applied CA in more projects

🐹 2020: Introduced CA in Go projects

🚀 Today: A core principle in most of my projects

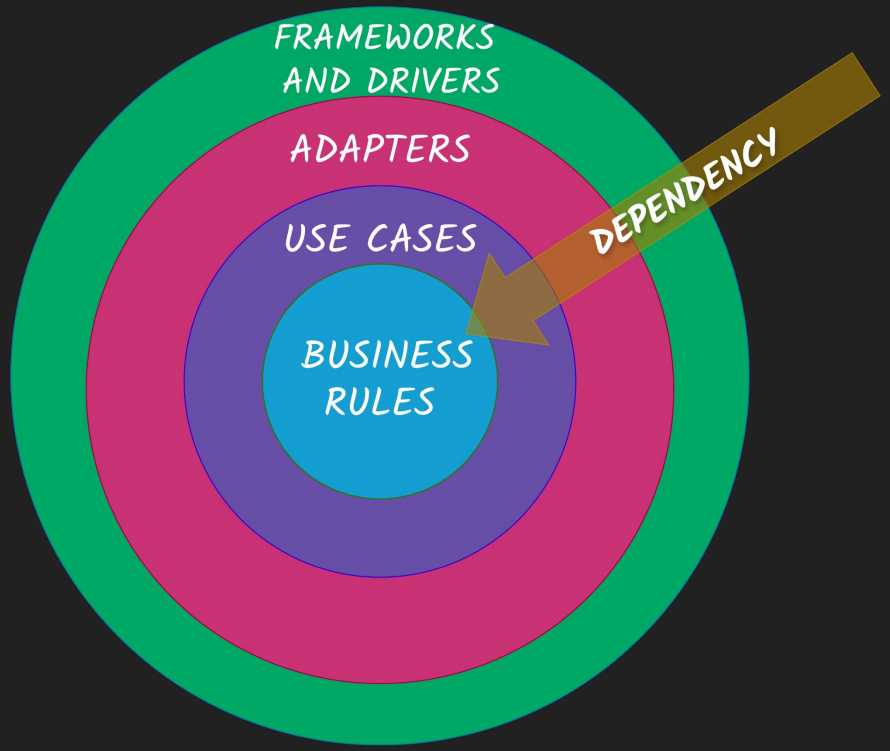
> whoami

- 👨‍💻 Panayiotis Kritiotis
- 🏢 Senior Engineering Manager @efood
- 🏗️ Software & Architecture Design, DDD, Leadership
- ✍️ Blogging at pkritiotis.io
- 🏃🧗 Running, Rock climbing
- 👧👦 Father of two objectively beautiful twins

What is Clean Architecture

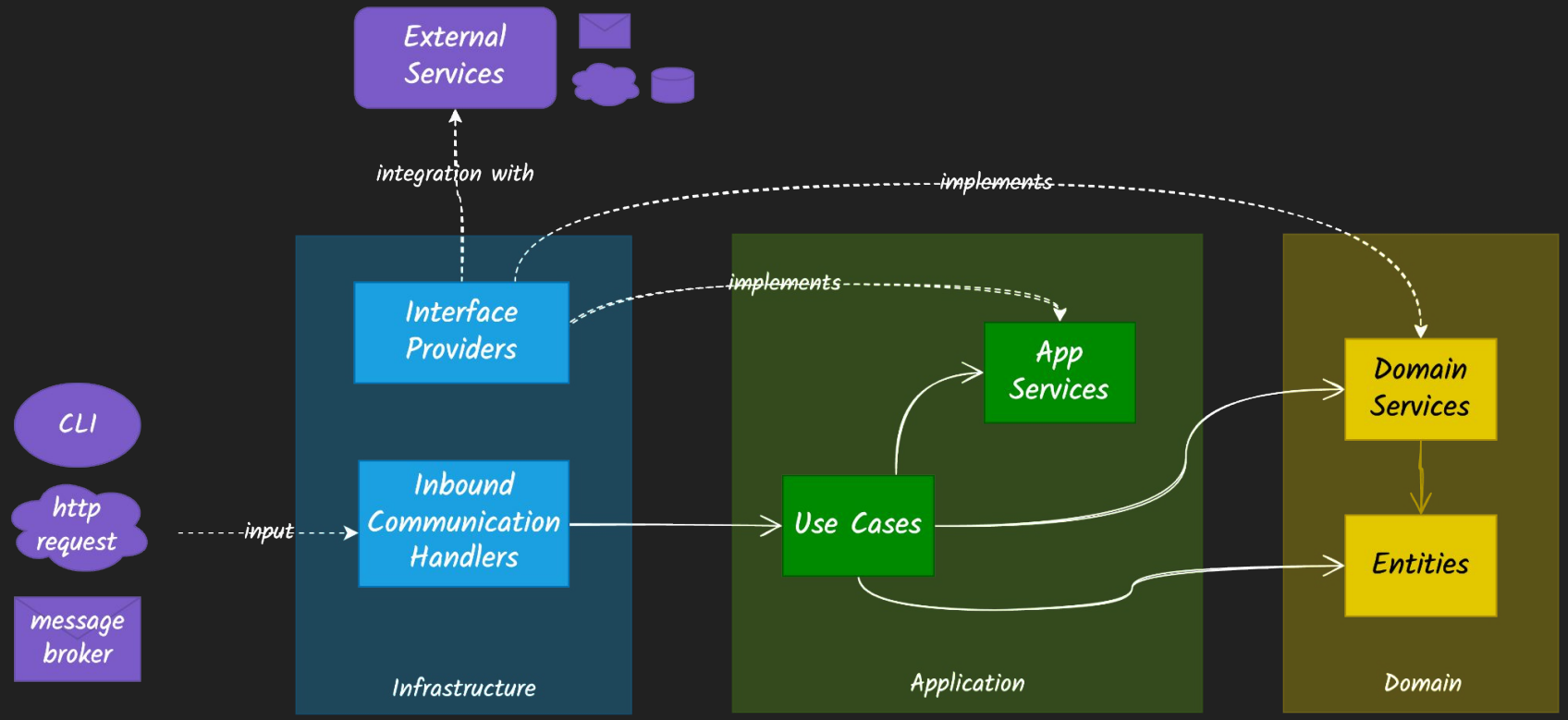
Software Design Philosophy that makes code Modular, Testable, and Maintainable by enforcing clear Separation of Concerns and Dependency Flow.

The Core Idea



The core business logic remains untouched by frameworks, databases, or external details.

A Three Layer version



Example: Race Tracking Service

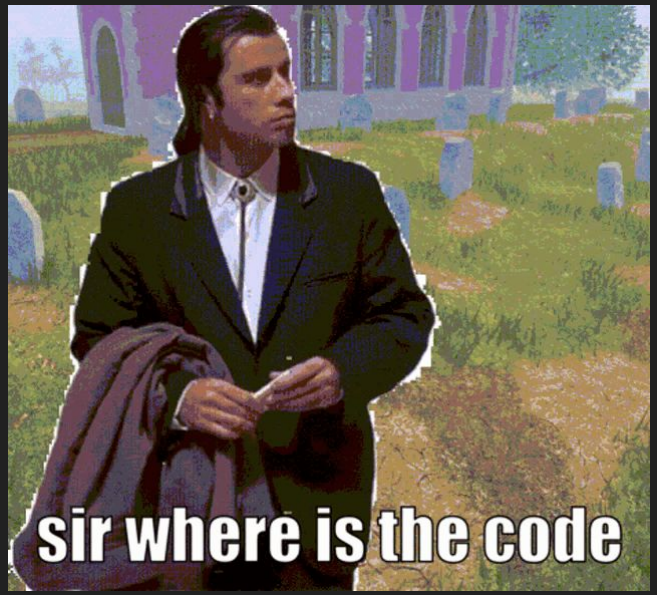
Domain (Entities)

- **Runner**s 🏃 participate in **Race**s 🏁
- Their race details are tracked in a race **Result** 📊

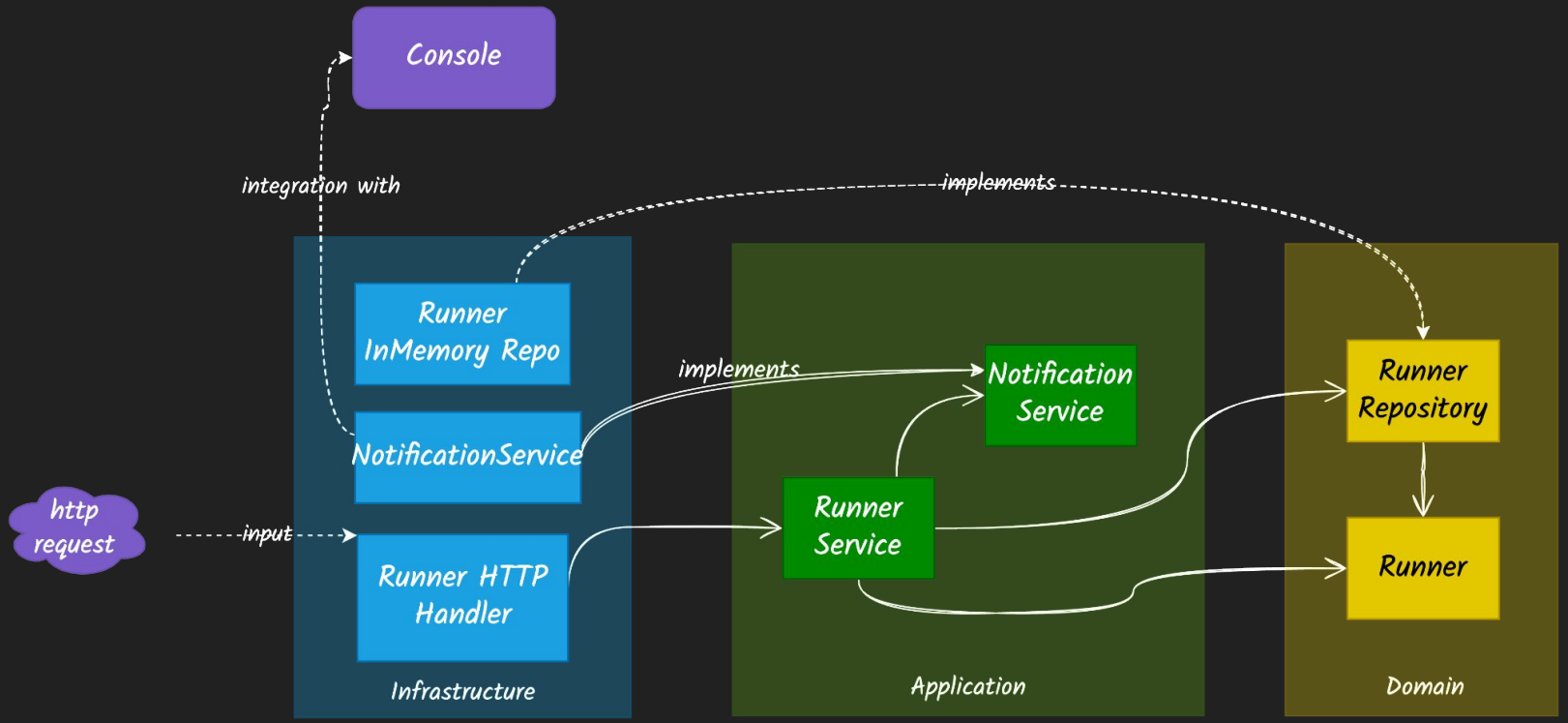
Features (Use Cases)

1. Register a **Runner** and send a notification on success
2. Create a **Race**
3. Log race **Result**s of a **Runner** for a specific **Race**
4. Return race results for a **Runner**

Show me
the code!



How we'll organize our code



Common Antipatterns

Antipattern #1 - Group by type

```
.
├── entities
│   ├── race.go
│   └── runner.go
├── factories
│   ├── race.go
│   └── runner.go
├── http
│   ├── race.go
│   └── runner.go
├── interfaces
│   ├── auth.go
│   ├── mapprovider.go
│   └── notification.go
├── models
│   ├── race.go
│   └── runner.go
├── mysql
│   ├── race.go
│   └── runner.go
├── notification
├── repos
│   ├── race.go
│   └── runner.go
├── services
│   ├── race.go
│   └── runner.go
└── valueobjects
    └── email.go
```

- ❌ Secure Layer Boundaries
- ❌ Domain Boundaries
- ❌ Idiomatic Packages

Antipattern #1 - Group by layer then by type

```
.
├── app
│   ├── interfaces
│   │   ├── auth.go
│   │   ├── mapprovider.go
│   │   └── notification.go
│   ├── models
│   │   ├── race.go
│   │   └── runner.go
│   └── services
│       ├── race.go
│       └── runner.go
├── domain
│   ├── entities
│   │   ├── race.go
│   │   └── runner.go
│   ├── factories
│   │   ├── race.go
│   │   └── runner.go
│   ├── repos
│   │   ├── race.go
│   │   └── runner.go
│   └── valueobjects
│       └── email.go
└── infra
    ├── http
    │   ├── race.go
    │   └── runner.go
    ├── mysql
    │   ├── race.go
    │   └── runner.go
    └── notification
```

✅ Secure Layer Boundaries

❌ Domain Boundaries

❌ Idiomatic Packages

Antipattern #1 - Group by layer

```
.
├── app
│   ├── race.go
│   └── runner.go
├── domain
│   ├── race.go
│   └── runner.go
└── infra
    ├── http.go
    ├── notification.go
    └── repositories.go
```

✅ Secure Layer Boundaries

❌ Domain Boundaries

❌ Idiomatic Packages

Solution #1 - Group by feature aka Vertical Slicing

```
.
├── race
│   ├── entity.go
│   ├── http.go
│   ├── repository.go
│   └── service.go
└── runner
    ├── entity.go
    ├── http.go
    ├── repository.go
    └── service.go
```

❌ Secure Layer Boundaries

✅ Domain Boundaries

✅ Idiomatic Packages

Solution #2 - Group by feature then by layer

```
├── race
│   ├── app
│   │   └── service.go
│   ├── domain
│   │   ├── race.go
│   │   └── result.go
│   └── infra
│       ├── http
│       │   └── handler.go
│       └── sql
│           └── repo.go
└── runner
    ├── app
    │   └── service.go
    ├── domain
    │   ├── email.go
    │   └── runner.go
    └── infra
        ├── http
        │   └── handler.go
        └── sql
            └── repo.go
```

✅ Secure Layer Boundaries

✅ Domain Boundaries

❌ Idiomatic Packages

⭐ Good for modular monoliths & bounded context isolation

Solution #3 - Group by layer then by feature

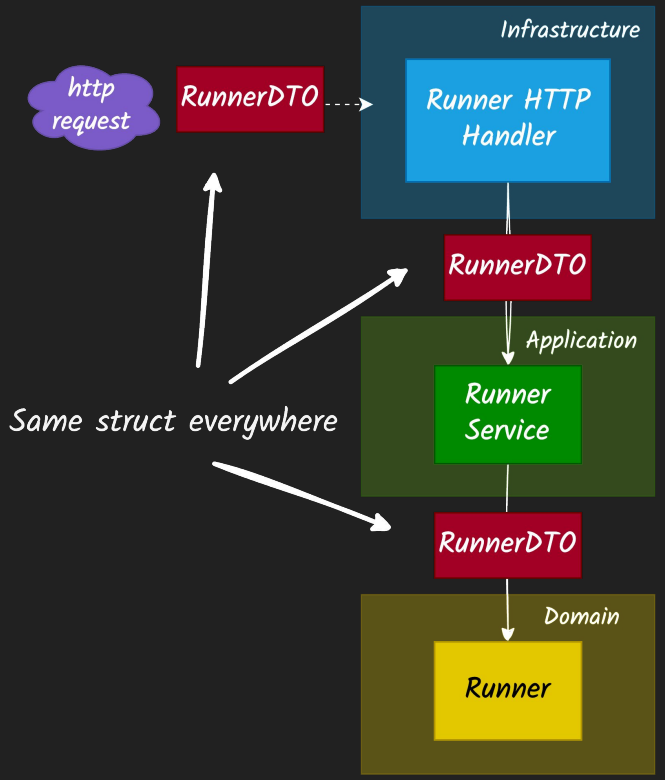
```
├── app
│   ├── bootstrap.go
│   ├── notification
│   │   ├── mock_notification.go
│   │   └── notification.go
│   ├── race
│   │   ├── service.go
│   │   └── service_test.go
│   └── runner
│       ├── service.go
│       └── service_test.go
├── domain
│   ├── race
│   │   ├── race.go
│   │   ├── race_test.go
│   │   ├── repository.go
│   │   ├── result.go
│   │   └── result_test.go
│   └── runner
│       ├── email.go
│       ├── repository.go
│       ├── runner.go
│       └── runner_test.go
└── infra
    ├── bootstrap.go
    ├── http
    │   ├── race
    │   │   ├── handler.go
    │   │   └── handler_test.go
    │   ├── runner
    │   │   ├── handler.go
    │   │   └── handler_test.go
    │   └── server.go
    ├── notification
    │   └── console
    │       ├── notificationservice.go
    │       └── notificationservice_test.go
    └── storage
        ├── memory
        │   ├── race
        │   │   ├── repository.go
        │   │   └── repository_test.go
        │   └── runner
        │       ├── repo.go
```

✅ Secure Layer Boundaries

✅ Domain Boundaries

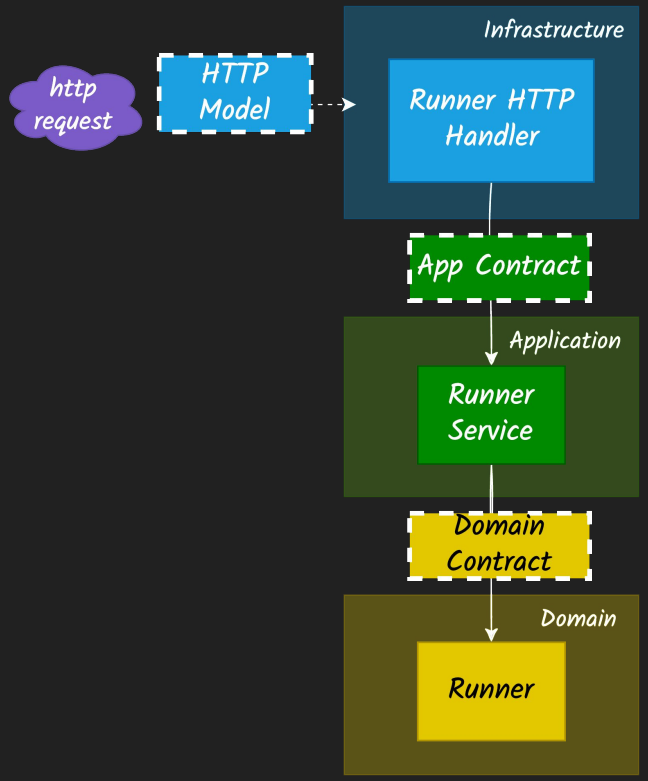
✅ Idiomatic Packages

Antipattern #2 - Model leaking



- ❌ Tight Coupling
- ❌ Business Logic Pollution
- ❌ Undefined State
- ❌ Unmaintainable

Antipattern #2 Solution - Data Contracts



- ✅ Low Coupling
- ✅ Layer Isolation
- ✅ Least Knowledge Principle

Antipattern #3 - Anemic Modeling

```go
type Runner struct {
    ID           uuid.UUID
    Name         string
    EmailAddress string
    CreatedAt    time.Time
}
```

"Here's the domain entity, do whatever you want"

❌ No behaviour

❌ No state protection

Key Principles

1. Domain → Protect your invariants
2. App layer → Use cases
3. Infra → Platform-specific implementation
4. Unit testing for Domain & App – Integration Testing for Infra
5. Respect the Data boundaries and the Dependency flow between layers

Beware of the Guy Who Only Talks About the Good

- Initial complexity and learning curve for newcomers
- More upfront effort - reduced speed in the early stages
- Verbose
- Not always necessary
- Potential to over-engineer

The Resistance
aka
Dirty Architects

The Resistance aka “The Dirty Gophers”

1. ~~Layers for the sake of organization~~ → Modularity
2. ~~Only for huge codebases~~ → Complexity-driven
3. ~~Overengineering~~ → Methodical Approach
4. ~~Only for OOP~~ → Any programing paradigm
5. ~~Unnecessary interfaces~~ → It’s called Testability

Final takeaways

#1

You need a design philosophy

Clean Architecture is a pretty good one

#2

Pragmatism beats dogma

Respect the philosophy,
Be flexible on the how

#3

Give Clean Architecture a try,

and give it a chance

Any Questions?