How do you structure your go apps?

Golang Bristol
24 July 2018

Kat Zień

Hello!

@kasiazien (https://twitter.com/kasiazien)

Questions, decisions

- Should I put everything in the main package?
- Should I start with one package and extract other packages over time?
- Should I use a framework?
- Can I use DDD?
- Microservices or monolith?
- How much should be shared?

Why should we care?

"Because if Go is going to be a language that companies invest in for the long term, the maintenance of Go programs, the ease of which they can change, will be a key factor in their decision." - Dave Cheney, Golang UK 2016 keynote

How do you get to a good structure?

Good structure goals

- Consistent.
- Easy to understand, navigate and reason about. ("makes sense")
- Easy to change, loosely-coupled.
- Easy to test.
- "As simple as possible, but no simpler."
- Design reflects exactly how the software works.
- Structure reflects the design exactly.

Demo project: a beer reviewing service

- Users can add a beer.
- Users can add a review for a beer.
- Users can list all beers or a specific beer.
- Users can list all reviews for a given beer.
- Option to store data either in memory or in a JSON file.
- Ability to add some sample data.

(for simplicity we'll skip deleting, updating and some error handling 😱)

Create a beer reviewing service

- beer
- review
- storage: memory and JSON file
- API
- sample data

Explore your options

Flat structure

```
/
├── Gopkg.toml
├── data.go
├── handlers.go
├── handlers_test.go
├── main.go
├── model.go
├── storage.go
├── storage_json.go
├── storage_json_test.go
├── storage_mem.go
├── storage_mem_test.go
└── storage_test.go
```

Group by function ("layered architecture")

- presentation / user interface
- application
- domain
- infrastructure

Group by module

Group by context

Domain Driven Development (DDD)
+ Hexagonal Architecture or the Actor Model

DDD

- Establish your domain and business logic.
- Define your bounded context(s), the models within each context and the ubiquituous language.
- Categorising the building blocks of your system:

Entity
Value Object
Domain Event
Aggregate
Service
Repository
Factory

Back to beer reviews

Context: an HTTP API for adding beer reviews

Language: beer, review, beer repository, ...

Models:

Entities: HTTP Server
Value Object: Beer, Review
Domain Event: Beer already exists, Beer not found (defined as errors in the demo app for simplicity and probably not ideal*)

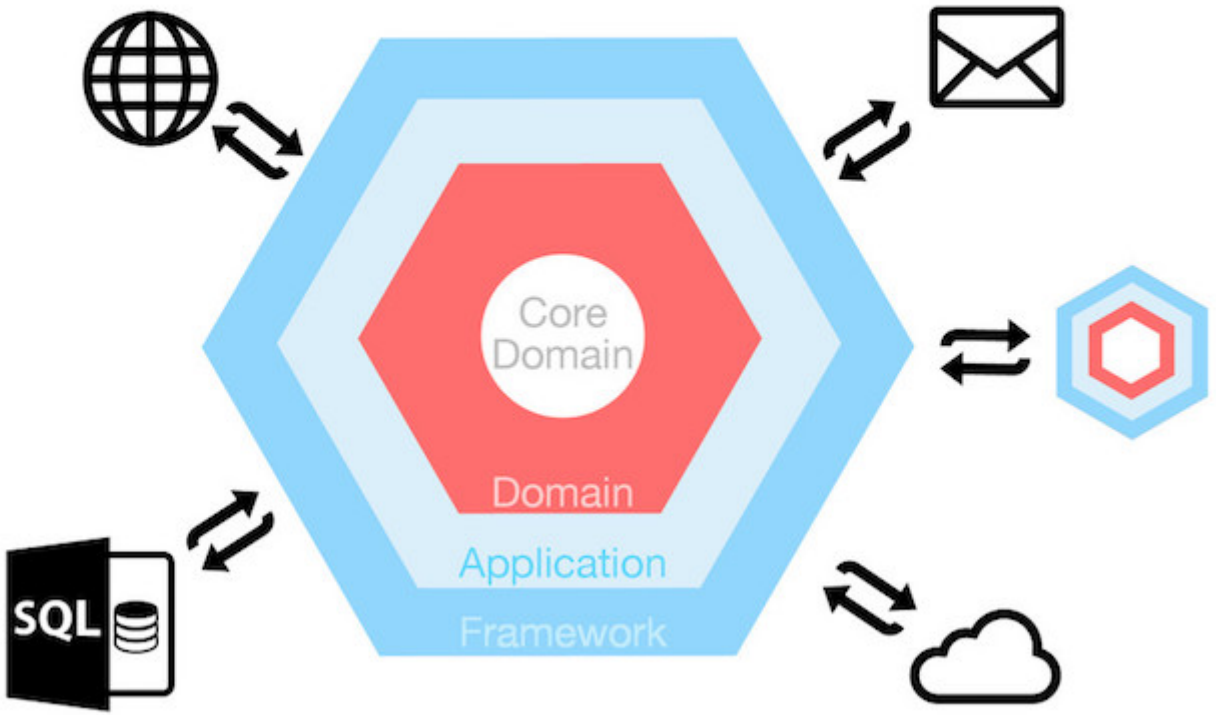
*https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully

Aggregates: Beer(s) adder, Review adder, Beer(s) lister, Review lister
Service: Add Beer, Add Review, List Beer, List Beers, List Beer Reviews
Repository: Beer Repository, Review Repository
Factories: Beer Adder, Review Adder (omitted in the demo app for simplicity)

Hexagonal architecture

- "ports and adapters"

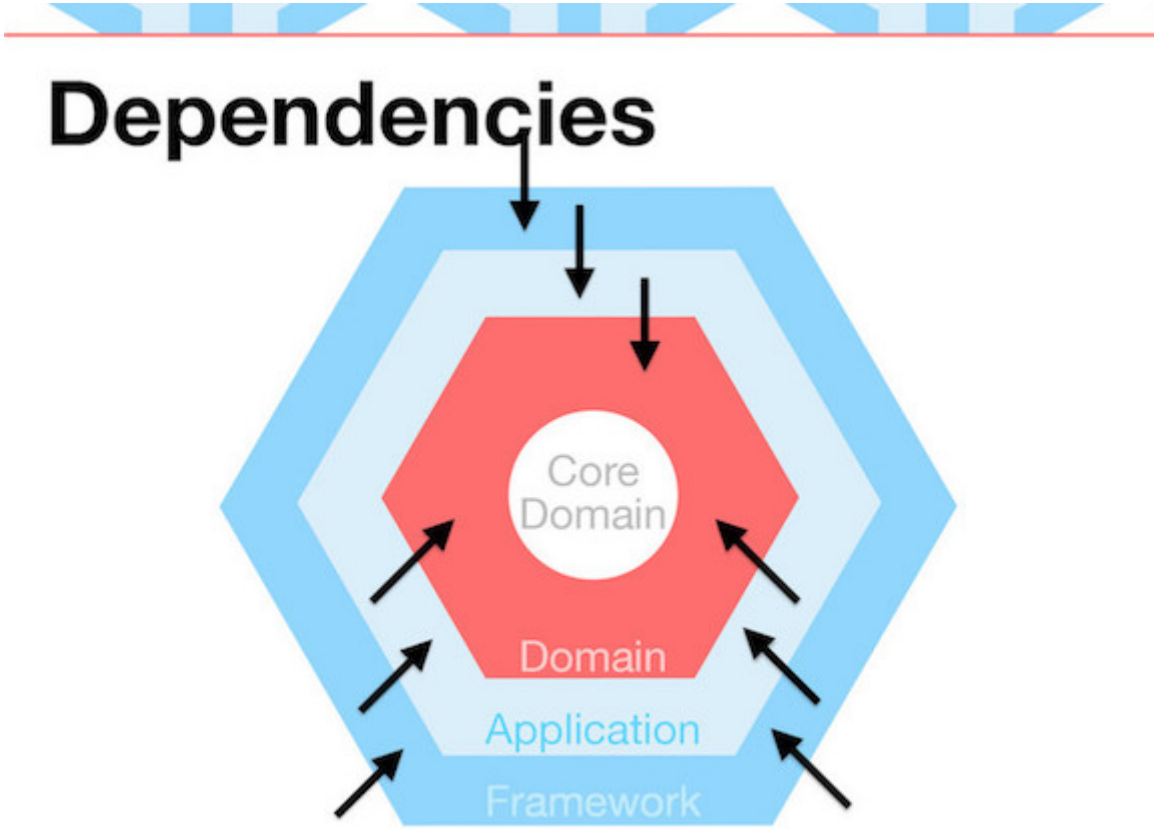
The Hexagon



source: http://fideloper.com/hexagonal-architecture

Hexagonal architecture

- Dependencies **only point inwards.**



source: http://fideloper.com/hexagonal-architecture

Putting it all together in Go

☝🏻 Two top-level directories: cmd (for each of your binaries) and pkg (for all your codez).

✌🏿 All other project files (build, dependencies, Docker, ...): root dir of your project.

🤟🏻 Domain types: root package.

🖐🏻 Dependencies: subpackages.

🖐 Mocks: shared subpackage.

🙌🏿 Main package initialises and ties everything together.

👎🏻 Avoid global variables and init().

https://peter.bourgon.org/blog/2017/06/09/theory-of-modern-go.html

https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1

Shortcut?

https://github.com/thockin/go-build-template (https://github.com/thockin/go-build-template)

The Actor model

- Each object is an actor with a mailbox and behaviour, with messages exchanged between actors through the mailboxes.
- Aggregates act as actors.
- All communication is performed asynchronously and without shared state between the actors.
- "Well suited for DDD and highly-scalable systems and potentially simpler to implement than a typical event-driven architecture."
- No shared state == potential use of concurrency? :)

Naming

- Choose package names that suggest well what can be expected inside.
- Avoid generic names like util, common etc.
- Follow the usual go conventions.

see https://talks.golang.org/2014/names.slide

- Remember that exported names are qualified by their package names, so avoid stutter if you can (e.g. strings.Reader not strings.StringReader).

Testing

- Keep the _test.go files next to the main files.
- Use a shared mock subpackage.

Judge your design

- Look at the exiting common ways of structuring projects.
- Prototype.
- Be like water.
- Good choices and best practices will come with experience.

Conclusion

- No single right answer 🙃
- Group code into packages by context rather than functional type.
- Avoid global scope for better maintainability.
- Separate code from project files and the main binaries.
- Mainatain consistency.

Questions?

@kasiazien

code: https://github.com/katzien/go-structure-examples (https://github.com/katzien/go-structure-examples)

references:

Go and a Package Focused Design, Gopher Academy Blog (https://blog.gopheracademy.com/advent-2016/go-and-package-focused-design/)

Standard Package Layout by Ben Johnson (https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1)

Repository structure by Peter Bourgon (http://peter.bourgon.org/go-best-practices-2016/#repository-structure)

Building an enterprise service in Go by Marcus Olsson (https://www.youtube.com/watch?v=twcDf_Y2gXY)

Hexagonal architecture by Chris Fidao (http://fideloper.com/hexagonal-architecture)

Thank you

Kat Zień
@kasiazien (http://twitter.com/kasiazien)