How do you structure your go apps?

GopherCon UK
3 August 2018

Kat Zień

Questions, decisions 🤔

- Should I put everything in the main package?
- Should I start with one package and extract other packages over time?
- How do I decide if something should be in its own package?
- Should I just use a framework?
- What's the programming paradigm for Go?
- Microservices or monolith?
- How much should be shared between packages?

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

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

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.
- 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 and tests 😱)

Flat structure

Flat structure

```
/
├── data.go
├── handlers.go
├── main.go
├── model.go
├── storage.go
├── storage_json.go
└── storage_mem.go
```

Group by function

Group by function ("layered architecture")

- presentation / user interface
- business logic
- external dependencies / infrastructure

Group by module

Group by context

Create a beer reviewing service

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

Context: an HTTP API for adding beer reviews

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

Entities: HTTP Server

Service: Add Beer, Add Review, List Beers, List Reviews

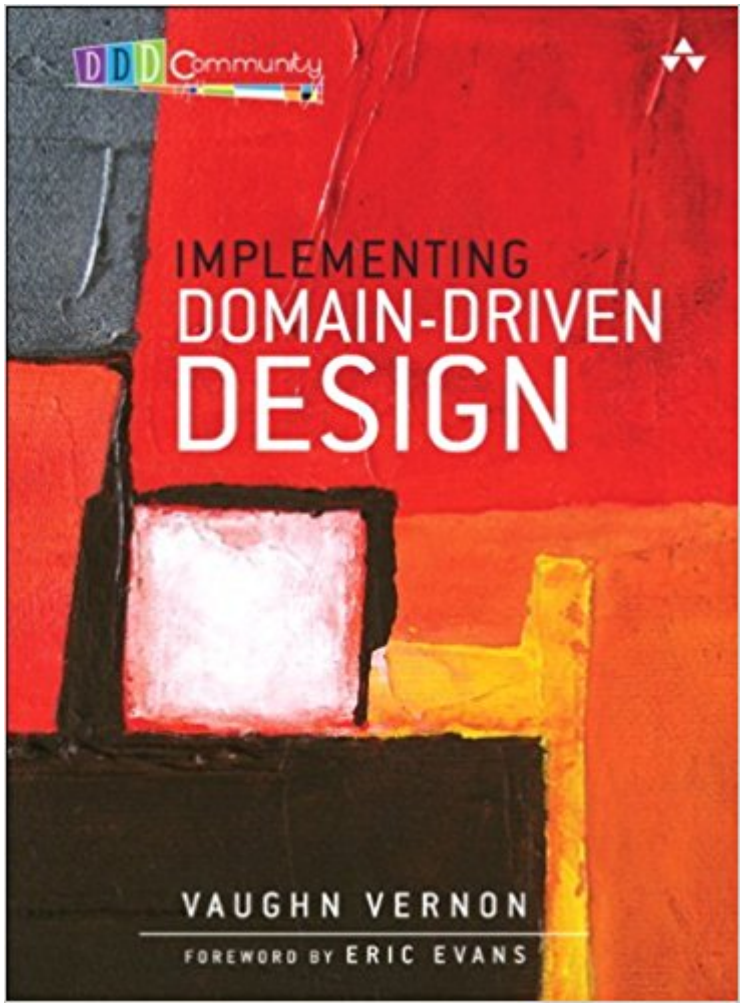
Value Objects: Beer, Review

Possible events: Beer already exists, Beer not found

Repository: Beer Repository, Review Repository

Aggregates: Beer adder, Review adder, Beer lister, Review lister

Domain Driven Development! (DDD)

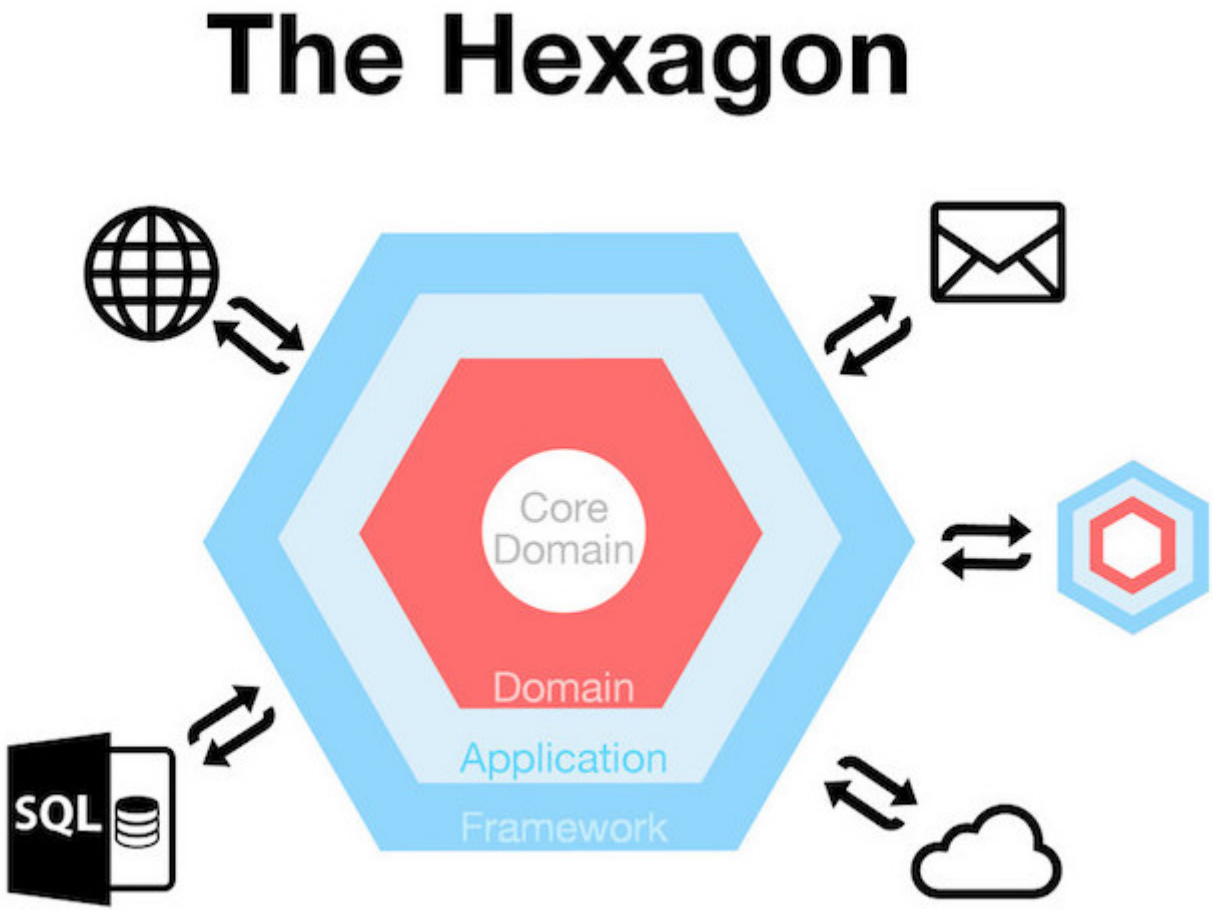


Domain Driven Development! (DDD)

- 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

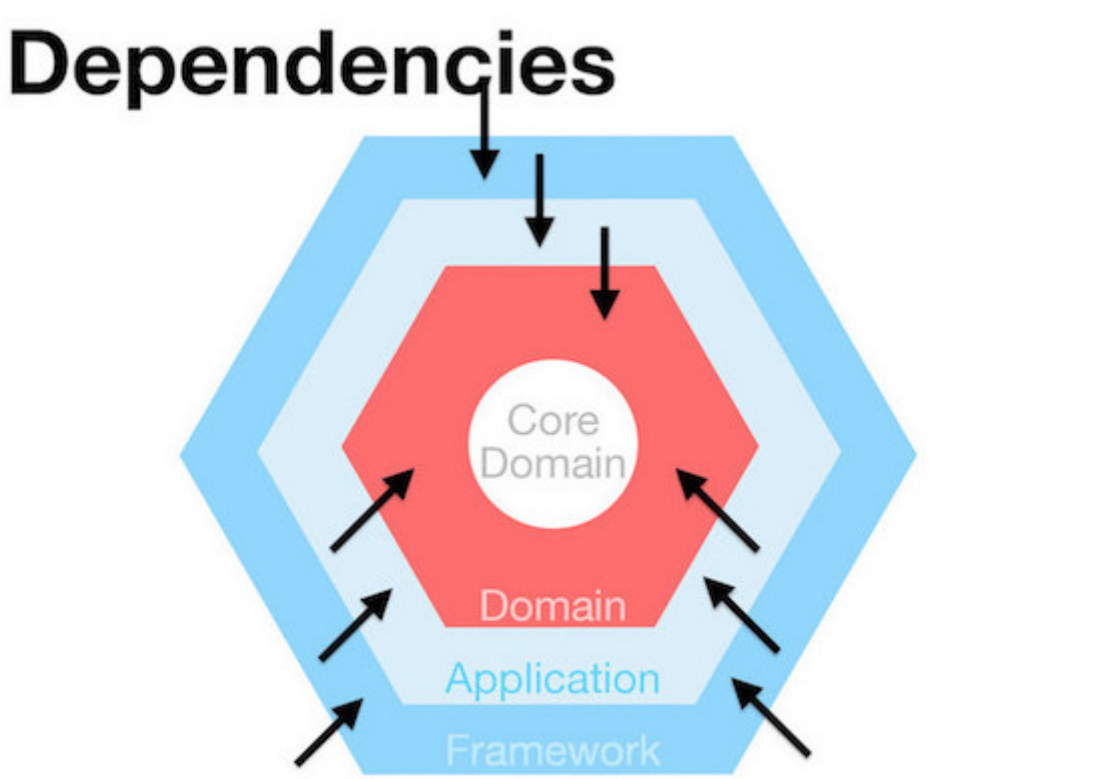
Hexagonal architecture ("ports and adapters")



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

Hexagonal architecture

- Core domain surrounded by interfaces to external inputs and outputs.
- Dependencies **only point inwards**.



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

The Actor Model

A model that treats "actors" as the universal primitives of concurrent computation.

In response to a message that it receives, an actor can:
- make local decisions
- create more actors
- send more messages
- determine how to respond to the next message received

Actors may modify their own private state, but can only affect each other through messages (avoiding the need for any locks).

from wikipedia

The Actor Model

No shared state... concurrency! 🚀

"Well suited for DDD and highly-scalable systems and potentially simpler to implement than a typical event-driven architecture." - Vaughn Vernon

Testing

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

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

- Avoid stutter (e.g. strings.Reader not strings.StringReader).

Frameworks?

intelligentbee.com/2017/08/14/golang-guide-list-top-golang-frameworks-ides-tools/

(https://intelligentbee.com/2017/08/14/golang-guide-list-top-golang-frameworks-ides-tools/)

Shortcut for rolling on your own?

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

Putting it all together in Go

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

✌🏾 Domain types: inside the root package.

🤟🏼 Dependencies: own packages.

🤚🏻 Mocks: shared subpackage.

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

🙌🏿 Main package initialises and ties everything together.

👎🏻 Avoid global scope and init().

(Huge) credit to:

Peter Bourgon

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

Ben Johnson

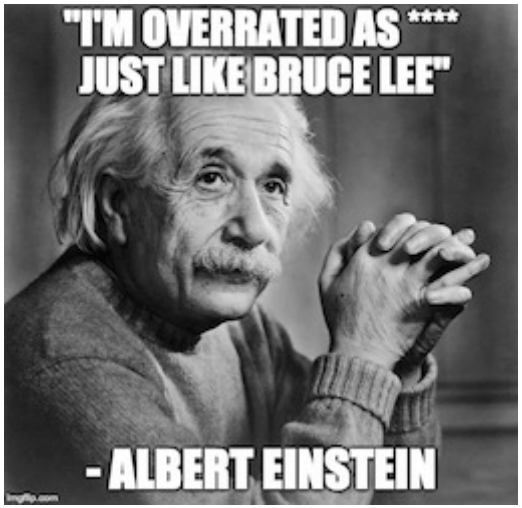
medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1 (https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1)

Marcus Olsson

github.com/marcusolsson/goddd (https://github.com/marcusolsson/goddd)

Conclusion

- No single right answer (sorry...) 🙃
- "Be like water"
- "As simple as possible, but no simpler"
- Experiment!
- Mainatain consistency
- Share your ideas 🙌🏼



Questions? Links!

@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)