Possible ways to design flexible APIs in Go

Session 19

Golang course by Exadel

19 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ Package structure recap
- ▶️ “Accept interface, return concrete type” rule
- ▶️ Examine the process of *publishing* open-source Go library
- ▶️ Real-life example project
- ▶️ API design challenges
- ▶️ Four options to make public API flexible

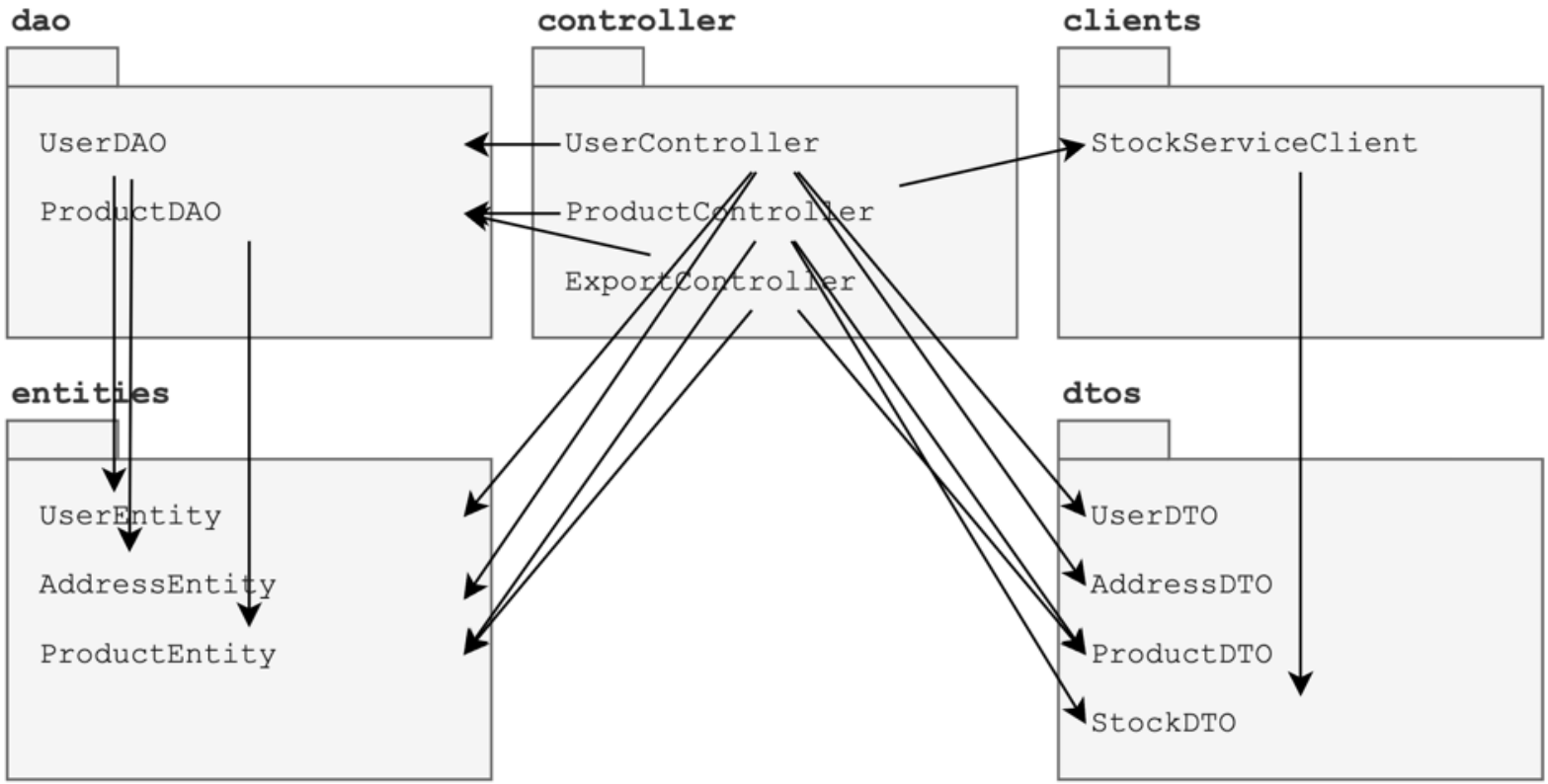
Package structure recap

▶️ For your project/library **start small**: maybe with just one `main.go` file and then grow your project gracefully

▶️ **Layer-based** architecture or **feature-based** architecture:

- Layered-based example:
  - "github.com/bxcodec/go-clean-arch" (https://github.com/bxcodec/go-clean-arch)
- Feature-based example:
  - "Package by Feature" (https://phauer.com/2020/package-by-feature/)
  - "Package by Layer vs Package by Feature" (https://medium.com/sahibinden-technology/package-by-layer-vs-package-by-feature-7e89cde2ae3a)

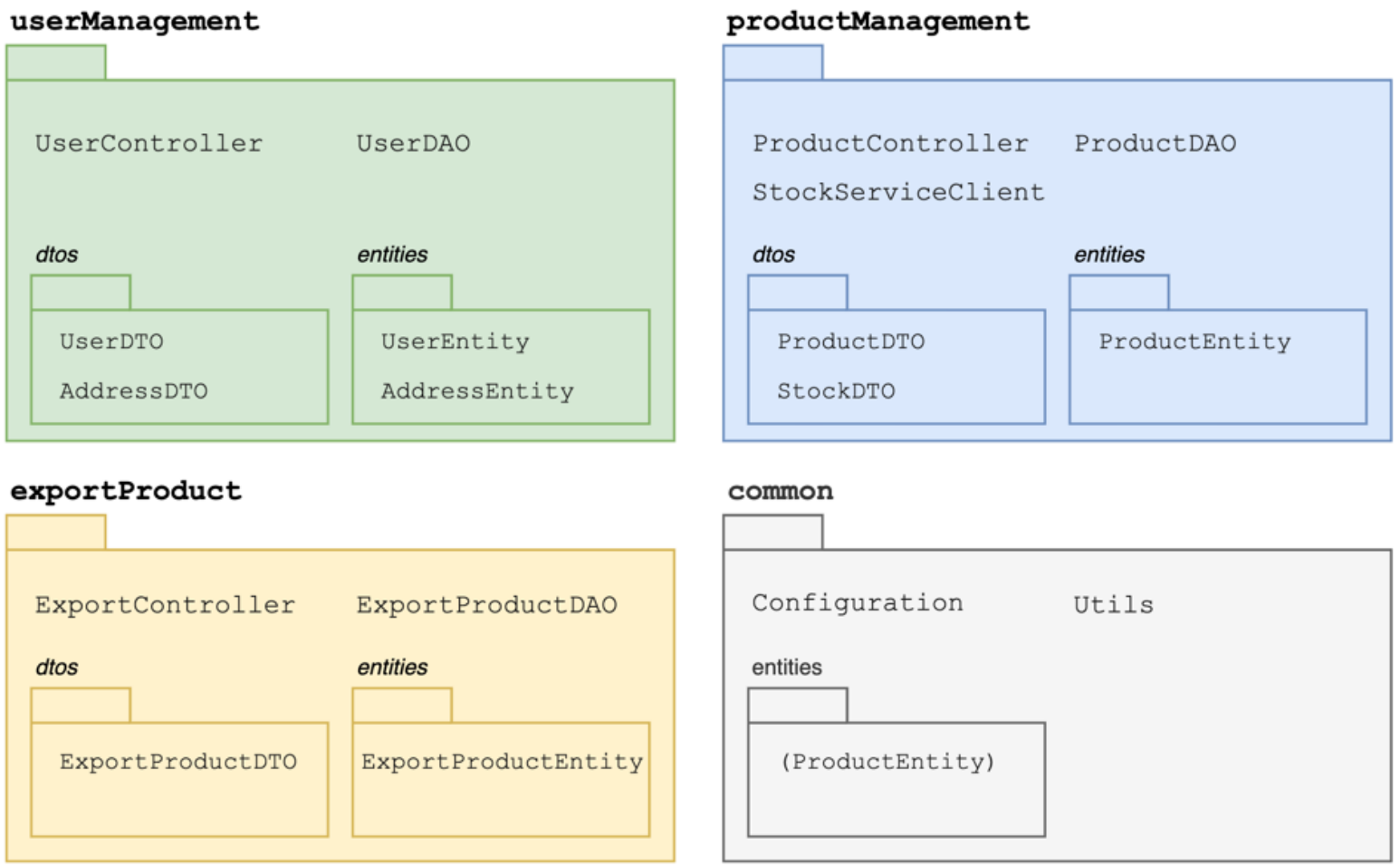
Layered-based example (in Kotlin, but the language doesn't matter):



The call hierarchy is spread across the whole project and involves many packages

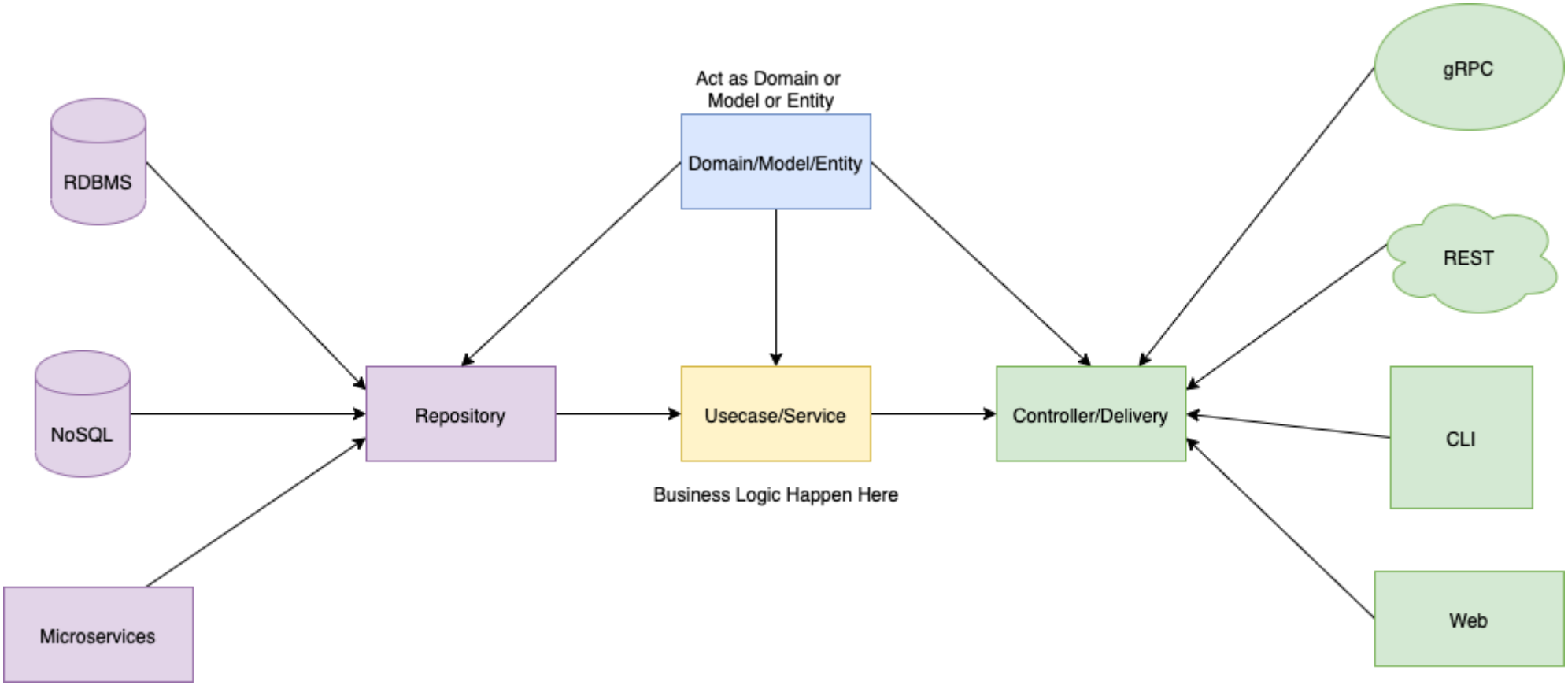
Source: "https://phauer.com/2020/package-by-feature/" (https://phauer.com/2020/package-by-feature/)

Feature-based example (in Kotlin, but the language doesn't matter):



Source: "https://phauer.com/2020/package-by-feature/" (https://phauer.com/2020/package-by-feature/)

Package decomposition example



Source: "https://github.com/bxcodec/go-clean-arch" (https://github.com/bxcodec/go-clean-arch)

“Accept interface, return concrete type” rule

▶️ Do not introduce interfaces "just because" :) Avoid interface pollution issue.

- Always `[abstract]` things when you actually **need them**, never when you just **foresee** that you need them.

▶️ Go interfaces shall be introduced on the "**API Consumer**" side and not on the "**API Producer**" side.

▶️ Go interfaces to abstract behavior. **Non**-interface types to describe data.

Source: "What “accept interfaces, return structs” means in Go" by Jack Lindamood

(https://medium.com/@cep21/what-accept-interfaces-return-structs-means-in-go-2fe879e25ee8)

For the rest of the presentation we need a real-life example code

- `graterm` - provides primitives to perform ordered **GRA** ceful **TERM** ination (aka shutdown) in Go application.
- Sourcecode - https://github.com/skovtunenko/graterm (https://github.com/skovtunenko/graterm)
- In general, here is the list of great libraries: https://awesome-go.com/ (https://awesome-go.com/)

Examine the source code 😎

👍 The purpose of the lib 😂

👍 **Test** coverage + goroutine leak tests

👍 **README.md** front page

👍 **Labels** aka Badges

👍 SemVer release **tags**, then later on add a section about API stability

👍 **GoDoc** for all Public methods/functions, comments in code

👍 Reference to other go types in the documentation

👍 **Examples** for methods and full-fledged sample application

👍 Establish external code coverage reports integration: https://codecov.io/gh/skovtunenko/graterm (https://codecov.io/gh/skovtunenko/graterm)

👍 Establish CI process with GitHub: https://github.com/skovtunenko/graterm/actions (https://github.com/skovtunenko/graterm/actions)

Think about errors vs logging

▶️ Most likely, the **logging** shall not be a part of the library

- There is no single standard logger interface in the Go ecosystem 🤔

▶️ To log more or to log less? Not clear 🤔

▶️ Possible solutions:

- By default, do not log anything, provide ability to inject custom logger with simple interface
- Use only errors, assume the library will work just fine in other cases
- Think about lifecycle inject points and events. Then let the API consumer to provide a callback to react on them

▶️ Do **NOT** log error and return it from the function/method!

Think about defaults

▶️ Most probably, understanding of **defaults** will be different for different people and use-cases

▶️ Explicit default functions/methods/consts/vars will bloat your public API (and hard to change them later)

▶️ Implicit defaults might be useful anyway

Think of release strategy and SemVer

- ▶️ Go packages with `v0.X.X` and `v1.X.X` are special
- ▶️ Versions like `v0.X.X` has no API guarantees
- ▶️ For versions like `v1.X.X` the API shall be stable - no breaking changes
- ▶️ Versions `v2.X.X+` shall change import path to add `.../v2` at the end
- ▶️ If you have a library, then update `README.md` to state your API guarantees

Collect feedback/review results

▶️ Code review results

▶️ Collect feedback, don't take it personally

▶️ Evaluate each comment/feedback: can you adjust your documentation to answer those questions?

- Example is here: Add more documentation for WithName()method and Hook type #74 (https://github.com/skovtunenko/graterm/issues/74)

API design challenges

Generic Advices

▶️ Prefer sync API over async API (if that's possible)

▶️ Do not store `context.Context` variables inside structs

- the only thing that can be stored is `cancel()` function to cancel context

▶️ For resource cleanup better to return a cleanup function explicitly:

```
package context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc) {.....}
```

▶️ Prefer anonymous `function` parameters to be at *the end of function* definition:

```
func (t *Terminator) Register(param1 Type1, param2 Type2, fn func(ctx context.Context) { }
```

- Then consumers can call the API like this:

```
t.Register(Order(1), func(ctx context.Context) {
    // some code
})
```

Complex things

▶️ Carefully craft the package/module boundaries

▶️ Find the best analogy for domain objects - steal from real world

- Example from `graterm` library: Maybe `Stopper`? Or `Terminator` ? Or `Shutdown`? Or `Halt` ?

▶️ Resist to add many features => revisit everything once the work is over => wipe out all unnecessary things

▶️ Ideally, APIs should be easy to use and hard to misuse (c) Josh Bloch

- Check `graterm`'s godoc for Hook (https://pkg.go.dev/github.com/skovtunenko/graterm#Hook) type.

▶️ The documentation/godoc of your library/module/package is a part of **public API** (*same guarantees expected!*)

Four options to make public API flexible

- ▶️ "rigid" API where all the function arguments are explicitly mentioned
- "rigid" API with vararg list in the end
- ▶️ Accept only mandatory arguments and optional arguments define as `config` struct
- ▶️ Functional options as per Dave Chaney article: "Functional options for friendly APIs" (https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis)
- ▶️ Fluent builders

Rigid API

- Example from graterm@v0.1.0 (https://github.com/skovtunenko/graterm/blob/v0.1.0/term.go#L92-L102):

```
// Register registers termination hook with priority and human-readable name.
// The lower the order the higher the execution priority, the earlier it will be executed.
// If there are multiple hooks with the same order they will be executed in parallel.
func (t *Terminator) Register(order Order,
                              componentName string,
                              timeout time.Duration,
                              hookFunc func(ctx context.Context)) {
    // ......
}
```

- We can't **change explicit function parameters** later on without breaking public API
- No way to make some parameters optional - *not flexible approach*

Rigid API with varargs

▶️ Same as before, but optional arguments will be expressed in form of varargs.

- Example: gorm's Find() method (https://pkg.go.dev/gorm.io/gorm#DB.Find)

Explicit params with optional `config`

▶️ Example:

```
type Config struct {
    componentName string
    timeout       time.Duration
    hookFunc      func(ctx context.Context)
}

func (t *Terminator) Register(order Order, config Config) {
    // .....
}
```

▶️ We can add more fields later into `Config` struct later on

- But new fields might support "default" behavior to support existing code

Functional options (by Dave Chaney)

```
type Option func(hook *Hook)

func WithName(name string) Option {
    return func(hook *Hook) { hook.name = name }
}

func WithHook(timeout time.Duration, hookFunc func(ctx context.Context)) Option {
    return func(hook *Hook) { hook.timeout = timeout; hook.hookFunc = hookFunc }
}

func (t *Terminator) Register(order Order, opts ...Option) {
    h := &Hook{}
    for _, opt := range opts {
        opt(h)
    }
    // now register the Hook inside Terminator
}

func Example() {
    t, _ := NewWithSignals(context.Background())
    t.Register(Order(1), WithName("name"), WithHook(5*time.Second, func(ctx context.Context) { /* .... */ }))
}
```

Source: "Functional options for friendly APIs" by Dave Chaney (https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis)

Builders

- Example from graterm@v0.1.0 (https://github.com/skovtunenko/graterm/blob/v0.1.0/term.go#L92-L102):

```
func NewWithSignals(appCtx context.Context, sig ...os.Signal) (*Terminator, context.Context) {...}

func (t *Terminator) WithOrder(order Order) *Hook {...}

func (h *Hook) WithName(name string) *Hook {...}

func (h *Hook) Register(timeout time.Duration, hookFunc func(ctx context.Context)) {...}
```

That's it. Thank you!

Thank you

Golang course by Exadel

19 Dec 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)