Project Structure

Idioms and suggestions from the Go community

Colton J. McCurdy

McCurdyColton

Detroit Go Meetup

November 19th, 2019

Credit Due

- GoTime.fm Ep. 102 - Application Design
- How to Structure Go Apps - Kat Zien

Motivation

In general, why is project structure important?

- Building a mental model / Readability
- Standardization
  - Reduce project on-boarding costs
  - Logging, monitoring and alerting
- Helps with maintenance costs
- Help manage dependencies
  - Specific and non-specific to Go
  - In Go, this is a compilation error
  - This was actually the motivation for creating Go

Motivation

In general, why is project structure important?

- Building a mental model / Readability
- Standardization
  - Reduce project on-boarding costs
  - Logging, monitoring and alerting
- Helps with maintenance costs
- Help manage dependencies
  - Specific and non-specific to Go
  - In Go, this is a compilation error
  - This was actually the motivation for creating Go

Motivation

In general, why is project structure important?

- Building a mental model / Readability
- Standardization
    - Reduce project on-boarding costs
    - Logging, monitoring and alerting
- Helps with maintenance costs
- Help manage dependencies
    - Specific and non-specific to Go
    - In Go, this is a compilation error
    - This was actually the motivation for creating Go

Motivation

In general, why is project structure important?

- Building a mental model / Readability
- Standardization
  - Reduce project on-boarding costs
  - Logging, monitoring and alerting
- Helps with maintenance costs
- Help manage dependencies
  - Specific and non-specific to Go
  - In Go, this is a compilation error
  - This was actually the motivation for creating Go

Motivation

In general, why is project structure important?

- Building a mental model / Readability
- Standardization
  - Reduce project on-boarding costs
  - Logging, monitoring and alerting
- Helps with maintenance costs
- Help manage dependencies
  - Specific and non-specific to Go
  - In Go, this is a compilation error
  - This was actually the motivation for creating Go

Motivation

Ultimately, **speed**

Now and in the **future**

Motivation

Ultimately, **speed**

Now and in the **future**

Consider

Before spending months on design, consider:

Context

Consider

Before spending months on design, consider:

- What problem(s) are you trying to solve?
- Will the project grow? How will it grow?
- Lifetime?
    - Of the **problem** and the project
    - Product-market fit?
- Who are your users?
    - Open-source library?
    - Public API for your company?
    - Internal tool or API at your company?
- How many users?
    - Library for Kubernetes?

Consider

Before spending months on design, consider:

- What problem(s) are you trying to solve?
- Will the project grow? How will it grow?
- Lifetime?
  - Of the **problem** and the project
  - Product-market fit?
- Who are your users?
  - Open-source library?
  - Public API for your company?
  - Internal tool or API at your company?
- How many users?
  - Library for Kubernetes?

Consider

Before spending months on design, consider:

- What problem(s) are you trying to solve?
- Will the project grow? How will it grow?
- Lifetime?
  - Of the **problem** and the project
  - Product-market fit?
- Who are your users?
  - Open-source library?
  - Public API for your company?
  - Internal tool or API at your company?
- How many users?
  - Library for Kubernetes?

Consider

Before spending months on design, consider:

- What problem(s) are you trying to solve?
- Will the project grow? How will it grow?
- Lifetime?
  - Of the **problem** and the project
  - Product-market fit?
- Who are your users?
  - Open-source library?
  - Public API for your company?
  - Internal tool or API at your company?
- How many users?
  - Library for Kubernetes?

Consider

Before spending months on design, consider:

- What problem(s) are you trying to solve?
- Will the project grow? How will it grow?
- Lifetime?
  - Of the **problem** and the project
  - Product-market fit?
- Who are your users?
  - Open-source library?
  - Public API for your company?
  - Internal tool or API at your company?
- How many users?
  - Library for Kubernetes?

Consider

Before spending months on design, consider:

Design **importance fluctuates** based on the context.

Standardization

Standardize or should leave experimentation up to teams?

- Context
  - How many teams?
  - How many repositories?
    - single-digits? tens? thousands?
- For adoption, having a standard in place is necessary
  - Define the “paved path”
- Can’t deviate from the standard creates barriers
  - Very few people making improvements

Don’t let standardization prevent innovation.

Standardization

Standardize or should leave experimentation up to teams?

- Context
  - How many teams?
  - How many repositories?
    - single-digits? tens? thousands?
- For adoption, having a standard in place is necessary
  - Define the “paved path”
- Can’t deviate from the standard creates barriers
  - Very few people making improvements

Don’t let standardization prevent innovation.

Standardization

Standardize or should leave experimentation up to teams?

- Context
  - How many teams?
  - How many repositories?
    - single-digits? tens? thousands?
- For adoption, having a standard in place is necessary
  - Define the “paved path”
- Can’t deviate from the standard creates barriers
  - Very few people making improvements

Don’t let standardization prevent innovation.

Standardization

Standardize or should leave experimentation up to teams?

- Context
    - How many teams?
    - How many repositories?
        - single-digits? tens? thousands?
- For adoption, having a standard in place is necessary
    - Define the "paved path"
- Can't deviate from the standard creates barriers
    - Very few people making improvements

Don't let standardization prevent innovation.

Standardization

Standardize or should leave experimentation up to teams?

- Context
  - How many teams?
  - How many repositories?
    - single-digits? tens? thousands?
- For adoption, having a standard in place is necessary
  - Define the “paved path”
- Can’t deviate from the standard creates barriers
  - Very few people making improvements

Don’t let standardization prevent innovation.

Remember

(If you **remember** one slide, this should be it)

- Structure / **abstractions will emerge**
- Rewrites are fine and often necessary
- Organizations and technologies will change
  - This will render your abstraction as useless
  - Or will make updating technologies difficult
  - Conway's Law
    - Organizations design systems that mirror their own communication structure

Solve the problem; design will emerge and often change

Remember

(If you **remember** one slide, this should be it)

- Structure / **abstractions will emerge**
- Rewrites are fine and often necessary
- Organizations and technologies will change
  - This will render your abstraction as useless
  - Or will make updating technologies difficult
  - Conway's Law
    - Organizations design systems that mirror their own communication structure

Solve the problem; design will emerge and often change

Remember

(If you **remember** one slide, this should be it)

- Structure / **abstractions will emerge**
- Rewrites are fine and often necessary
- Organizations and technologies will change
  - This will render your abstraction as useless
  - Or will make updating technologies difficult
  - Conway's Law
    - Organizations design systems that mirror their own communication structure

Solve the problem; design will emerge and often change

Remember

(If you **remember** one slide, this should be it)

- Structure / **abstractions will emerge**
- Rewrites are fine and often necessary
- Organizations and technologies will change
  - This will render your abstraction as useless
  - Or will make updating technologies difficult
  - Conway's Law
    - Organizations design systems that mirror their own communication structure

Solve the problem; design will emerge and often change

Remember

(If you **remember** one slide, this should be it)

- Structure / **abstractions will emerge**
- Rewrites are fine and often necessary
- Organizations and technologies will change
  - This will render your abstraction as useless
  - Or will make updating technologies difficult
  - Conway's Law
    - Organizations design systems that mirror their own communication structure

Solve the problem; design will emerge and often change

Go Background

```
  pkg/
    a/
      a.go # package a
    b/
      b.go # package b
```

```
$ cat pkg/a/a.go
package a
import "b"
```

```
$ cat pkg/b/b.go
package b
import "a"
```

Go Background

```
  pkg/
    a/
      a.go # package a
    b/
      b.go # package b
```

```
$ cat pkg/a/a.go
package a
import "b"
```

```
$ cat pkg/b/b.go
package b
import "a" <---- "import cycle not allowed"
```

Go Background

- Appreciate the “import cycle not allowed” error
- I fought this error a lot when I started, but I rarely see it now
- If you’re fighting this error, consider a redesign, refactor or simplifying
- Dependency management — packages are dependencies — is important

Rob Pike comparing compilation times from C++ to Go

“…turns minutes into seconds, coffee breaks into interactive builds” – Rob Pike at SPLASH 2012

Patterns

- No wrong “solution”, just possibly better “solutions”
- “Bad” abstractions are worse than no abstractions
- **Understand the flow of requests through packages**
- **Part of learning is discovering what doesn’t work**

Patterns

- No wrong “solution”, just possibly better “solutions”
- “Bad” abstractions are worse than no abstractions
- Understand the flow of requests through packages
- Part of learning is discovering what doesn’t work

Patterns

- No wrong “solution”, just possibly better “solutions”
- “Bad” abstractions are worse than no abstractions
- **Understand the flow of requests through packages**
- **Part of learning is discovering what doesn’t work**

Patterns

- No wrong “solution”, just possibly better “solutions”
- “Bad” abstractions are worse than no abstractions
- **Understand the flow of requests through packages**
- **Part of learning is discovering what doesn’t work**

Patterns

Where do I put ...

- Tests
  - **No** `tests/`
  - `name_test.go` files remain in the package with the related `name.go` file
- `cmd/`
  - Multiple binaries / “entrypoints”
- `internal/` vs `pkg/`
  - `internal/` – “ensures that changes to the API of internal packages will never break an external application”
- Where do I put everything else?
  - `Dockerfile`, `README.md`, dotfiles, etc.

Abstractions

What are we trying to solve with abstractions?

- Efficient mental model building
- Readability
- Reduce maintenance costs
- **Ultimately, speed**

Don’t abstract just to abstract

Patterns

1. Flat Structure (i.e., "abstractionless")

- This is a great starting place
- No package abstractions
- Everything is in `package main`
    - No "import cycle" errors

Patterns

1. Flat Structure (i.e., "abstractionless")

- This is a great starting place
- No package abstractions
- Everything is in `package main`
  - No "import cycle" errors

Patterns

1. Flat Structure (i.e., “abstractionless”)

- This is a great starting place
- No package abstractions
- Everything is in `package main`
    - No “import cycle” errors

Patterns

1. Flat Structure (i.e., “abstractionless”)

```
main.go
server.go
database.go
thing1.go # model, view and controller code
thing1_test.go
thing2.go # model, view and controller code
thing2_test.go
```

Patterns

1. Flat Structure (i.e., “abstractionless”)

Challenges:

- Mental model construction is difficult from project structure alone
  - Ineffective display of “grouping”, layering and request flow
- Readability

These become more true as the **project grows in size**.

Patterns

1. Flat Structure (i.e., “abstractionless”)

Challenges:

- Mental model construction is difficult from project structure alone
    - Ineffective display of “grouping”, layering and request flow
- Readability

These become more true as the **project grows in size**.

Patterns

1. Flat Structure (i.e., “abstractionless”)

Challenges:

- Mental model construction is difficult from project structure alone
  - Ineffective display of “grouping”, layering and request flow
- Readability

These become more true as the **project grows in size**.

Patterns

1. Flat Structure (i.e., “abstractionless”)

Challenges:

- Mental model construction is difficult from project structure alone
    - Ineffective display of “grouping”, layering and request flow
- Readability

These become more true as the **project grows in size**.

Patterns

1. Flat Structure (i.e., “abstractionless”)

Benefits:

- Immediately tackling the problem(s) at hand
- Gives abstractions time to emerge; if they exist
- Easy to identify and build abstractions from this point

Patterns

1. Flat Structure (i.e., “abstractionless”)

Benefits:

- Immediately tackling the problem(s) at hand
- Gives abstractions time to emerge; if they exist
- Easy to identify and build abstractions from this point

Patterns

1. Flat Structure (i.e., “abstractionless”)

Benefits:

- Immediately tackling the problem(s) at hand
- Gives abstractions time to emerge; if they exist
- Easy to identify and build abstractions from this point

Patterns

1. Flat Structure (i.e., “abstractionless”)

Benefits:

- Immediately tackling the problem(s) at hand
- Gives abstractions time to emerge; if they exist
- Easy to identify and build abstractions from this point

Patterns

2. Model-View-Controller (MVC)

```
main.go
pkg/
  controllers/ # package controllers
    thing1.go
    thing2.go
  database/ # package database
    database.go
  models/ # package models
    thing1.go
    thing2.go
  views/ # package views
    thing1.go
    thing2.go
```

Patterns

2. Model-View-Controller (MVC)

Challenges:

- To do well, requires you to use Go `interfaces`
  - If you are new to Go, this could be a challenge
- Code duplication to avoid circular dependencies
  - You will most likely have a model and response for the same type that are tightly-coupled
  - Controller calls models and builds a view
- Related “things” are “**far**”

Patterns

2. Model-View-Controller (MVC)

Challenges:

- To do well, requires you to use Go `interfaces`
    - If you are new to Go, this could be a challenge
- Code duplication to avoid circular dependencies
    - You will most likely have a model and response for the same type that are tightly-coupled
    - Controller calls models and builds a view
- Related “things” are “**far**”

Patterns

2. Model-View-Controller (MVC)

Challenges:

- To do well, requires you to use Go `interfaces`
  - If you are new to Go, this could be a challenge
- Code duplication to avoid circular dependencies
  - You will most likely have a model and response for the same type that are tightly-coupled
  - Controller calls models and builds a view
- Related “things” are “**far**”

Patterns

2. Model-View-Controller (MVC)

Challenges:

- To do well, requires you to use Go `interfaces`
  - If you are new to Go, this could be a challenge
- Code duplication to avoid circular dependencies
  - You will most likely have a model and response for the same type that are tightly-coupled
  - Controller calls models and builds a view
- Related “things” are “**far**”

Patterns

2. Model-View-Controller (MVC)

Benefits:

- Centralized logic for interacting with a data store
  - Easier to swap technologies (e.g., PostgreSQL to MySQL), if you have abstracted the technology away from the model
- Standard outside of Go
- Related “things” are “**close**”

Patterns

2. Model-View-Controller (MVC)

Benefits:

- Centralized logic for interacting with a data store
  - Easier to swap technologies (e.g., PostgreSQL to MySQL), if you have abstracted the technology away from the model
- Standard outside of Go
- Related “things” are “**close**”

Patterns

2. Model-View-Controller (MVC)

Benefits:

- Centralized logic for interacting with a data store
  - Easier to swap technologies (e.g., PostgreSQL to MySQL), if you have abstracted the technology away from the model
- Standard outside of Go
- Related “things” are “**close**”

Patterns

2. Model-View-Controller (MVC)

Benefits:

- Centralized logic for interacting with a data store
  - Easier to swap technologies (e.g., PostgreSQL to MySQL), if you have abstracted the technology away from the model
- Standard outside of Go
- Related “things” are “**close**”

Patterns

Addition Patterns (I’m still learning how to apply these)

- Domain-driven design (DDD)
    - Similar goals to micro-services
    - Separating parts of the business
    - Domain-specific logic (i.e., for this service, let’s do retries)
- Hexagonal architecture

My Framework

How I learned (and continue to learn)

- Go Package-focused design
- Ben Johnson's blog posts
    - Standard Package Layout
    - Structuring Applications in Go
- `github.com/golang-standards/project-layout`
- Use a popular open-source example as a **reference** (don't just copy)
    - Kubernetes, Docker, Yay, FZF, HashiCorp/*, etc.
    - `github.com/trending/go?since=weekly`
    - Go's stdlib – `github.com/golang/go`

My Framework

How I learned (and continue to learn)

I failed (and still fail), a lot

Conclusion

There is no one “correct” design