Structuring Web Applications in Go: Previous Article Next Article

Moving Towards Domain Driven Design in Go

The goal of this article is to help illustrate how an application might evolve over time so that we can understand some of the problems that a more domain driven design might help with. To that end, we are going to look at a fairly trivial project as it evolves over time. This project won't be complete - the sample code won't compile, isn't tested with a compiler, and doesn't even list imports. It is simply meant to be an example to follow along with. That said, if anything seems wrong feel free to reach out and I'll fix it up or answer your questions (if I can!)

First, let's discuss the project. Imagine that you are at work and your boss asks you to create a way to authenticate users via the GitHub API. More specifically, you are going to be given a user's personal access token, and you need to look up the user as well as all of their organizations. That way you can later restrict their access based on what orgs they are a part of.

Note: We are using access tokens to simplify the examples.

Sounds easy enough, so you fire up your editor and whip up a `github` package that provides this functionality.

```
package github

type User struct {
  ID    string
  Email string
```

```
  Email string
  OrgIDs []string
}

type Client struct {
  Key   string
}

func (c *Client) User(token string) (User, error) {
  // ... interact with the github API, and return a user if they are in an org wit
}
```

Note: I am not really using the GitHub API here - this is a mostly made up example.

Next you to take your `github` package and write some middleware that can be used to protect some of our HTTP handlers. In this middleware you will retrieve a user's access token from a basic auth header and then use the GitHub code to look up the user, check to see if they are part of the provided org, then grant or deny access accordingly.

```
package mw

func AuthMiddleware(client *github.Client, reqOrgID string, next http.Handler) htt
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    token, _, ok := r.BasicAuth()
    if !ok {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    user, err := client.User(token)
    if err != nil {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    permit := false
    for _, orgID := range user.OrgIDs {
      if orgId == reqOrgID {
        permit = true
        break
      }
    }
```

```
    if !permit {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    // user is authenticated, let them in
    next.ServeHTTP(w, r)
  })
}
```

You present this to your peers and they are concerned with the lack of tests. More specifically, there doesn't seem to be a way to verify this `AuthMiddleware` works as advertised. "No problem," you say, "I'll just use an interface so we can test it!"

```
package mw

type UserService interface {
  User(token string) (github.User, error)
}

func AuthMiddleware(us UserService, reqOrgID string, next http.Handler) http.Handl
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    token, _, ok := r.BasicAuth()
    if !ok {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    user, err := us.User(token)
    if err != nil {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    permit := false
    for _, orgID := range user.OrgIDs {
      if orgId == reqOrgID {
        permit = true
        break
      }
    }
    if !permit {
```

```
    if !permit {
      http.Error(w, "Unauthorized", http.StatusUnauthorized)
      return
    }
    // user is authenticated, let them in
    next.ServeHTTP(w, r)
  })
}
```

Now you can test this code with a mock user service.

```
package mock

type UserService struct {
  UserFn func(token string) (github.User, error)
}

func (us *UserService) Authenticate(token string) (github.User, error) {
  return us.UserFn(token)
}
```

There are a bunch of different ways to create this mock user service, but this is a pretty common approach to test both cases where authentication works and cases where it returns an error.

The authentication middleware is then tested, released, and life seems jolly.

And then tragedy strikes. Your CEO hears that Godzilla is on his way to San Francisco and your company just can't keep using GitHub with that kind of uncertainty. What if their entire office is crushed and no engineers are around who know the product?!? Nope, that won't do. Completely unacceptable.

Luckily there is this alternative company named GitLab that seems to do a lot of the same things GitHub does, but they have a remote team. That means Godzilla can never wipe out all of their engineers, right? 🎉

The higher ups at your company seem to agree with this logic and they start to make the transition. Your job? You are tasked with making sure all of that authentication code you wrote works with the new system!

You spend some time looking at the GitLab API docs, and the good news is it looks like the same overall strategy will still work. GitLab has personal access tokens, organizations, and you just need to re-implement the client. The code in the middleware shouldn't need changed at all because you were a smart cookie and you used an INTERFACE! 😁 You get to work creating the GitLab client…

```
package gitlab


type User struct {
  ID    string
  Email string
  OrgIDs []string
}

type Client struct {
  Key   string
}

func (c *Client) User(token string) (User, error) {
  // ... interact with the gitlab API, and return a user if they are in an org wit
}
```

Then you go to plug that into the `AuthMiddleware`, but wait a minute, it won't work!

It turns out even interfaces can fall victim to coupling. In this case it is because your interface expects a `github.User` to be returned by the `User` method.

```
type UserService interface {
  User(token string) (github.User, error)
                      ^^^^^^^^^^^
}
```

What are you to do? Your boss wants this shipped yesterday!

At this point you have a few options:

1. Change your middleware so that the `UserService` interfaces expects a `gitlab.User` instead of a `github.User`

2. Create a new authentication middleware specifically for GitLab.
3. Create a common user type that will allow both your github and gitlab implementations to be interchangeable in the `AuthMiddleware`

(1) might make sense if you were confident that your company was going to stick with GitLab. Sure, you will need to change both the user service interface and the mocks, but if it is a one time change isn't so bad.

On the other hand, you don't really know that your org will stick with GitLab. After all, who let's Godzilla attacks dictate their decision process?

This option can also be problematic if many pieces of code throughout your application are reliant on the `github.User` type returned by this package.

(2) would work, but it seems a bit silly. Why would we want to rewrite ALL of that code and all of those tests when none of the logic is changing? Surely there must be a way to make this interface thing work as you originally intended. After all, that middleware really doesn't care how the user is looked up as long as we have a few critical pieces of information to work with.

So you decide to give (3) a shot. You will create a `User` type in your `mw` package and then you will write an adapter to connect it with the GitLab client you created.

```
package mw

type User struct {
  OrgIDs []string
}

type UserService interface {
  User(token string) (User, error)
}
```

As you write the code, you come to another realization; because you don't really care about things like a user's ID or email you can drop those fields entirely from your `mw.User` type. All you need to specify here are fields you actually care about, which should make things easier to maintain and test. Neato!

Next up you need to create an adapter, so you get to work on it.

```
// Package adapter probably isn't a great package name, but this is a
// demo so deal with it.
```

```
package adapter

type GitHubUserService struct {
  Client *github.Client
}

func (us *GitHubUserService) User(token string) (mw.User, error) {
  ghUser, err := us.Client.User(token)
  if err != nil {
    return mw.User{}, err
  }
  return mw.User{
    OrgIDs: ghUser.OrgIDs,
  }, nil
}

type GitLabUserService struct {
  Client *gitlab.Client
}

func (us *GitLabUserservice) User(token string) (mw.User, error) {
  glUser, err := us.Client.User(token)
  if err != nil {
    return mw.User{}, err
  }
  return mw.User{
    OrgIDs: glUser.OrgIDs,
  }, nil
}
```

You also need to update your mock, but that is a pretty quick change.

```
package mock

type UserService struct {
  UserFn func(token string) (mw.User, error)
}

func (us *UserService) Authenticate(token string) (mw.User, error) {
```

```
  return us.UserFn(token)
}
```

And now if you want to use our `AuthMiddleware` with either GitHub or GitLab you can do so with code something like this:

```
var myHandler http.Handler
var us mw.UserService
us = &GitLabUserservice{
  Client: &gitlab.Client{
    Key: "abc-123",
  },
}
// This protects your handler
myHandler = mw.AuthMiddleware(us, "my-org-id", myHandler)
```

Alas, we finally have a solution that is completely decoupled. We can easily switch between GitHub and GitLab, and when the new hip source control company kicks off we are prepared to hop on that bandwagon.

Finding a middle ground

In the previous example, we gradually watched the code go from what I consider tightly coupled to completely decoupled. We did this using the `adapter` package, which handles translating between these decoupled `mw` and `github/gitlab` packages.

The primary benefit is what we saw at the very end - we can now decide whether to use a GitHub or GitLab authentication strategy when setting up our handlers and our authentication middleware is entirely agnostic of which we choose.

While these benefits are pretty awesome, it isn't fair to explore these benefits without also exploring the costs. All of these changes presented more and more code, and if you look at the original version of out `gitlab` and `mw` packages they were significantly simpler than the final versions that need to make use of the `adapter` package. This final setup can also lead to more setup, as somewhere in our code we need to instantiate all of these adapters and plug things together.

If we continued down this route, we might quickly find that we need many different `User` typ
well. For example, we might need to associate an internal user type with external user IDs i
services like GitHub (or GitLab). This could lead to defining an `ExternalUser` in our datab

package and then writing an adapter to convert a `github.User` into this type so that our database code is agnostic to which service we are using.

I actually tried doing this on one project with my HTTP handlers just to see how it turned out. Specifically, I isolated every endpoint in my web application to its own package with no external dependencies specific to my web application and ended up with packages like this:

```
// Package enroll provides HTTP handlers for enrolling a user into a new
// course.
// This package is entirely for demonstrative purposes and hasn't been tested,
// but if you do see obvious bugs feel free to let me know and I'll address
// them.
package enroll

import (
  "io"
  "net/http"

  "github.com/gorilla/schema"
)

// Data defines the data that will be provided to the HTML template when it is
// rendered.
type Data struct {
  Form    Form
  // Map of form fields with errors and their error message
  Errors  map[string]string

  User    User
  License License
}

// License is used to show the user more info about what they are enrolling in.
// Eg if they URL query params have a valid key, we might show them:
//
//   "You are about to enroll in Gophercises - FREE using the key `abc-123`"
//                               ^            ^                  ^
//                           Course       Package              Key
//
type License struct {
  Key     string
```

```
    Key     string
    Course  string
    Package string
  }

  // User defines a user that can be enrolled in courses.
  type User struct {
    ID string
    // Email is used when rendering a navbar with the user's email address, among

    // other areas of an HTML page.
    Email string
    Avatar string
    // ...
  }

  // Form defines all of the HTML form fields. It assumes the Form will be
  // rendered using struct tags and a form package I created
  // (https://github.com/joncalhoun/form), but it isn't really mandatory as
  // long as the form field names match the `schema` part here.
  type Form struct {
    License  string `form:"name=license;label=License key;footer=You can find this i
  }

  // Handler provides GET and POST http.Handlers
  type Handler struct {
    // Interfaces and function types here serve roughly the same purpose. funcs
    // just tend to be easier to write adapters for since you don't need a
    // struct type with a method.
    UserFn    func(r *http.Request) (User, error)
    LicenseFn func(key string) (License, error)

    // Interface because this one is the least likely to need an adapter
    Enroller interface {
      Enroll(userID, licenseKey string) error
    }

    // Typically satisfied with an HTML template
    Executor interface {
      Execute(wr io.Writer, data interface{}) error
    }
  }
```

```
}

// Get handles rendering the Form for a user to enroll in a new course.
func (h *Handler) Get() http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    user, err := h.UserFn(r)
    if err != nil {
      // redirect or render an error
      return

    }
    var data Data
    data.User = user

    var form Form
    err := r.ParseForm()
    if err != nil {
      // maybe log this? We can still render
    }
    dec := schema.NewDecoder()
    dec.IgnoreUnknownKeys(true)
    err = dec.Decode(&form, r.Form)
    if err != nil {
      // maybe log this? We can still render
    }
    data.Form = form

    if form.License != "" {
      lic, err := h.LicenseFn(form.License)
      data.License = lic
      if err != nil {
        data.Errors = map[string]string{
          "license": "is not valid",
        }
      }
    }

    h.Executor.Execute(r, data)
  }
}

// Post handles processing the form and enrolling a user
```

```
// Post handles processing the form and enrolling a user.
func (h *Handler) Post() http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
    user, err := h.UserFn(r)
    if err != nil {
      // redirect or render an error
      return
    }
    var data Data

    data.User = user

    var form Form
    err := r.ParseForm()
    if err != nil {
      // maybe log this? We can still render
    }
    dec := schema.NewDecoder()
    dec.IgnoreUnknownKeys(true)
    err = dec.Decode(&form, r.Form)
    if err != nil {
      // maybe log this? We can still render
    }
    data.Form = form

    err = h.Enroller.Enroll(user.ID, form.License)
    if err != nil {
      data.Errors = map[string]string{
        "license": "is not valid",
      }
      // Re-render the form
      h.View.Execute(r, data)
      return
    }
    http.Redirect(w, r, "/courses", http.StatusFound)
  }
}
```

In theory this idea sounded pretty cool. I could now define all of my HTTP handlers in isolati without worrying about the rest of my application. Each package could be tested easily, and

I was writing these individual pieces I found myself incredibly productive. I even had an interface named `Executor`, and who doesn't want an executor in their code?!?

In practice, this idea was awful for my particular use case. Yes, there were benefits, but they weren't outweighing the cost of writing all this code. I was productive when creating the internals of the `enroll` and similar packages, but I spent so much time writing adapters and connecting pieces together that it crushed my productivity overall. I couldn't find a quick way to plug this into my code without needing to write a custom `UserFn`, `LicenseFn`, and I found myself writing a bunch of virtually identical variants of `UserFn` for every package with http handlers.

This leads to the topic of this section - **is there a way to come up with a reasonable middle ground?**

I like to decouple my code from third party dependencies. I like writing testable code. But I don't like doubling my coding efforts to make this happen. Surely there must be a middle ground here that gives us most of the benefits without all that extra code, right?

Yes, yes there is a middle ground and the key to finding it isn't to remove all coupling, but to intentionally pick and choose what your code is coupled to.

Let's go back to our original example with the `github` and `gitlab` packages. In our first version - the tightly coupled version - we had a `github.User` type that our `mw` package had a dependency on. It works well enough to get started, and we can even build interfaces around it, but we are still tightly coupled to the `github` package.

In our second version - the decoupled version - we had a `github.User`, a `gitlab.User`, and `mw.User`. This allowed us to decouple everything, but we have to create adapters that attach these decoupled pieces together.

The middle ground, and the third version we will explore, is to intentionally define a `User` type that every package is allowed to be tightly coupled to. By doing this, we are intentionally choosing where that coupling happens and can make that decision in a way that still makes it easy to test, swap implementations, and do everything else we desire from decoupled code.

First up is our `User` type. This will be created in a `domain` package that any other packages in our application can import.

```
package domain

type User struct {
  ID    string
  Email string
```

```
  OrgIDs []string
}
```

Next we will rewrite our `github` and `gitlab` packages to leverage this `domain.User` type. These are basically the same since I stubbed out all the real logic, so I'll only show one.

```
package gitlab // github is the same basically

type Client struct {
  Key   string
}

// Note the return type is domain.User here - this code is now coupled to our
// domain.
func (c *Client) User(token string) (domain.User, error) {
  // ... interact with the gitlab API, and return a user if they are in an org wit
}
```

And finally we have the `mw` package.

```
package mw

type UserService interface {
  User(token string) (domain.User, error)
}

func AuthMiddleware(us UserService, reqOrgID string, next http.Handler) http.Handl
  // unchanged
}
```

We can even write a mock package using this setup.

```
package mock

type UserService struct {
```

```
  UserFn func(token string) (domain.User, error)
}

func (us *UserService) Authenticate(token string) (domain.User, error) {
  return us.UserFn(token)
}
```

Domain Driven Design

I have tried to avoid any confusing terms up to this point because I find that they often complicate matters rather than simplify them. If you don't believe me, go try to read any articles, books, or other resources on domain driven design (DDD). They will almost always leave you with more questions and less clarity about how to actually implement the ideas in your code.

I'm not suggesting that DDD isn't useful, nor am I suggesting that you shouldn't ever read those books. What I am saying is that many (most?) of my readers are here looking for more practical advice on how to improve their code, not to discuss the theory of software development.

From a practical perspective, the key benefit of domain driven design is writing software that can evolve and change over time. The best way I have discovered to achieve this in Go is to clearly define your domain types, and to then write implementations that depend upon these types. This still results in coupled code, but because your domain is so tightly linked to the problem you are solving this coupling is rarely problematic. In fact I often find that needing a clear definition of domain models to be enlightening rather than troubling.

Note: This idea of defining concrete domain types and coupling code to them isn't unique or new. Ben Johnson wrote about it in 2016 and this is still an incredibly valuable article for any new Gopher.

Going back to the previous example, we saw our domain being defined in the `domain` package:

```
package domain

type User struct {
  ID    string
  Email string
  OrgIDs []string
}
```

Taking that a step further, we could even start to define basic building blocks that the rest of our application can either (a) implement, or (b) make use of without being coupled to implementations details. For instance in the case of our `UserService`:

```
package domain

type User struct { ... }

type UserService interface {
  User(token string) (User, error)
}
```

This is implemented by the `github` and `gitlab` packages, and relied upon by the `mw` package. It could also be relied upon by other packages in our code without worrying about how it gets implemented. And because it is defined at the domain, we don't need to worry about each implementation altering slightly in return types - they all have a common definition to build from.

As an application evolves and changes over time this idea of defining common interfaces to build from becomes even more powerful. For instance, imagine we had a `UserService` that is a little more complex; perhaps it handles creating users, authenticating users, looking them up via tokens, password reset tokens, changing passwords, and more.

```
package domain

type UserStore interface {
  Create(NewUser) (*User, RememberToken, error)
  Authenticate(email, pw string) (*User, RememberToken, error)
  ByToken(RememberToken) (*User, error)
  ResetToken(email string) (ResetToken, error)
  UpdatePw(pw string, tok ResetToken) (RememberToken, error)
}
```

We might start by implementing this with pure SQL code and a local database:

```
package sql

type UserStore struct {
  DB *sql.DB
```

```
}

func (us *UserStore) Create(newUser domain.NewUser) (*domain.User, domain.Remember
  // ...
}

// ... and more methods
```

This makes complete sense when we have a single application, but perhaps we start to grow into another Google and we decide that we need a centralized user management system that all of our separate applications can utilize.

If we had coupled our code to the `sql` implementation this might be challenging to achieve, but because most of our code is coupled to the `domain.UserService` we can just write a new implementation and use it instead.

```
package userapi


type UserStore struct {
  HTTPClient *http.Client
}

func (us *UserStore) Create(newUser domain.NewUser) (*domain.User, domain.Remember
  // interact with a third party API instead of a local SQL database
}

// ...
```

More generally speaking, coupling to a domain rather than a specific implementation allows us to stop worrying about details like:

- **Are we interacting with a microservice or a local database?** We can write code with reasonable timeouts regardless of whether our user management system is a local SQL database or a microservice.
- **Do we communicate with our user API via JSON, GraphQL, gRPC, or something else?** While our implementation will need to know how to communicate with the users API, the rest of will continue to operate the same regardless of which specific technology we are using.

- And much more…

At its crux, this is what I consider to be the primary benefit of domain driven design. It isn't fancy terms, colorful graphics, or looking smart in front of your peers. It is purely about designing software that is capable of evolving to meet your ever-changing requirements.

Why don't we just start here?

The obvious followup question at this point is, "Why didn't we just start with domain driven design if it is so great?"

Anyone with some experience using Model-View-Controller (MVC) can tell you that it is susceptible to the tight coupling. Nearly all of our application will need to depend on our models, and we just explored how that can be problematic. So what gives?

While building from a common domain can be useful, it can also be a nightmare if misused. Domain driven design has a fairly steep learning curve; not because the ideas are particularly hard to grasp, but because you rarely learn where you went wrong in applying them until a project grows to a reasonable size. As a result, it may take a few years until you really start to grasp all the dynamics involved. I have been writing software for quite a while, and I still don't feel like I have a full grasp on all the ways things can go wrong or get complicated.

Note: This is one of the big reasons why I have taken so long to publish this article. I was hesitant to share when, in many ways, I still don't feel like I am an expert on this topic. I ultimately decided to share because I believe others can learn from my limited understanding, and I believe this article can evolve and improve over time as I have discussions with other developers. So feel free to reach out to discuss it - jon@calhoun.io

MVC presents you with a reasonable starting point for organizing your code. Database interacts go here (models), http handlers go here (controllers), and rendering code goes here (views). It might lead to tight coupling, but it allows you to get started pretty quickly.

Unlike MVC, domain driven design doesn't present you with a reasonable starting point for how to organize your code. In fact, starting with DDD is pretty much the exact opposite of starting with MVC - rather than jumping right into building controllers and seeing how your models evolve, you instead have to spend a great deal of time upfront deciding what your domain should be. This likely involves mocking up some ideas and having peers review them, discussing what is/isn't right, a few iteration cycles, and only then can you dive into writing some code. You can see this in Ben Johnson's WTF Dial project where he creates a PR and discusses the domain with Peter Bourgon, Egon Elbre, and Marcus Olsson.

This isn't specifically a bad thing, but it also isn't easy to get right and it requires a great dea upfront work. As a result, I often find this working best if you have a larger team where ever

needs to agree on some common domain before development can start.

Given that I am often coding in smaller teams (or by myself), I find that my projects evolve much more naturally if I start with something simpler. Maybe that is a flat structure, maybe it is an mvc structure, or maybe it is something else entirely. I don't get too caught up in those details, as long as I am open to my code evolving. This allows it to eventually take the form of something like DDD, but it doesn't require me to start there. As I stated before, this may be harder to do with a larger org where everyone is developing the same application together, so more upfront design discussion is often merited.

In our sample application we did something very similar to this "let it evolve" concept. Every step was taken for a specific purpose; we added a `UserService` interface because we needed to test our authentication middleware. When we started to migrate from GitHub to GitLab we realized that our interface didn't suffice, so we explored alternative options. It is around that point that I think a more DDD approach starts to make sense and rather than guessing at what the `User` and `UserService` should look like, we have real implementations to base it off of.

Another potential issue with starting of with DDD is that types can be defined poorly because we are often defining them before we have concrete use cases. For instance, we might decide that authenticating a user looks like this:

```
type UserAuthenticator interface {
  Authenticate(email, pw string) (error)
}
```

Only later we might realize that in practice every single time we authenticate a user we really want to have the user (or maybe a remember token) returned, and that by defining this interface upfront we missed this detail. Now we need to either introduce a second method to retrieve this information, or we need to alter our `UserAuthenticator` type and refactor any code that implements or utilizes this.

The same thing applies to your models. Before actually implementing a `github` and `gitlab` package we might think that the only identifying information we need on a `User` model is an `Email` field, but we might later learn through implementing these services that an email address can change, and what we also need is an `ID` field to uniquely identify users.

Defining a domain model before using it is challenging. We are extremely unlikely to know what information we do and don't need unless we already have a very strong grasp of the domain we are working in. Yes, this might mean that we have to refactor code later, but doing so will be easier than refactoring your entire codebase because you defined a your domain incorrectly

is another reason why I don't mind starting with tightly-coupled code and refactoring at a later date.

Finally, not all code needs this sort of decoupling, it doesn't always provide the benefits it promises, and in some circumstances (eg DBs) we rarely take advantage of this decoupling.

For a project that isn't evolving, you likely don't need to spend all the time decoupling your code. If the code isn't evolving, changes are far less likely to occur and the extra effort of preparing for them may just be wasted effort.

Additionally, decoupling doesn't always provide the benefits it promises and we don't always take advantage of that decoupling. As Mat Ryer likes to point out, we very rarely just swap out our database implementation. And even if we do decouple everything, and even if we do happen to be in the very small minority of applications who are transitioning databases, this transition often requires a complete rethinking of how we interact with our data store; after all, a NoSQL database behaves completely differently from a SQL database and to really take advantage of either we have to write code that is specific to the database being used. The final result is that these abstractions don't always provide us with the magical, "the implementation doesn't matter" results that we want.

That doesn't mean DDD can't provide benefits, but it does mean we shouldn't simply drink the Kool-Aid and expect magical results. We need to stop and think for ourselves.

In Summary

In this article we looked firsthand at the problems encountered when code is tightly coupled, and we explored how defining domain types and interfaces can help improve this coupling. We also discussed some of the reasons why it might not be the best idea to start off with this decoupled design, and to instead let our code evolve over time.

In the next article in this series I hope to expand upon the idea of writing Go code using domain driven design. Specifically, I want to discuss:

- How interface tests can help ensure implementations can be interchanged without issue.
- How subdomains can also stem from different contexts.
- Ways you can visualize this all using the traditional DDD hexagon, as well as how code like a third party library might fit into the equation.

I also want to mention that this article is by no means a hard set of rules. It is just my meager attempt at sharing some insights and ideas that have helped me improve my Go software.

I'm also not the first to discuss or explore DDD and design patterns in Go. You should definitely check out some of the following for a more rounded understanding:

- GoDDD by Marcus Olsen - This is a GitHub repo along with a writeup and a talk where Marcus Olsen explores porting the traditional sample DDD Java app into Go.
- WTF Dial by Ben Johnson - I've already linked to Ben's Standard Pacakage Layout article; this is a series where Ben applies what he discusses in that article. I also recommend checking out the accompanying PRs and reading through the comments.
- How Do You Structure Your Go Apps by Kat Zien - In this talk Kat goes through quite a few ways to structure your Go apps. Also check out the repo and slides that accompany this talk.
- Design Philosophy On Packaging by Bill Kennedy - While not specifically about structuring your application, this series discusses package design which is tightly linked to structure.

This article is part of the series, Structuring Web Applications in Go. Previous Article Next Article

Recent Articles All Articles Mini-Series Progress Updates Tags

About Me Go Courses

Written by

Jon Calhoun

Follow @joncalhoun 4,786 followers

Jon Calhoun is a full stack web developer who also teaches about Go, web development, algorithms, and anything programming related. He also consults for other companies who have development needs. (If you need some development work done, get in touch!)

Jon is a co-founder of EasyPost, a shipping API that many fortune 500 companies use to power their shipping infrastructure, and prior to founding EasyPost he worked at google as a software engineer.

Jon's latest progress update: Writing Course Notes

More in this series

This post is part of the series, Structuring Web Applications in Go.

Spread the word

Did you find this page helpful? Let others know about it!

Tweet

Sharing helps me continue to create both free and premium Go resources.

Want to discuss the article?

See something that is wrong, think this article could be improved, or just want to say thanks? I'd love to hear what you have to say!

You can reach me via email or via twitter.