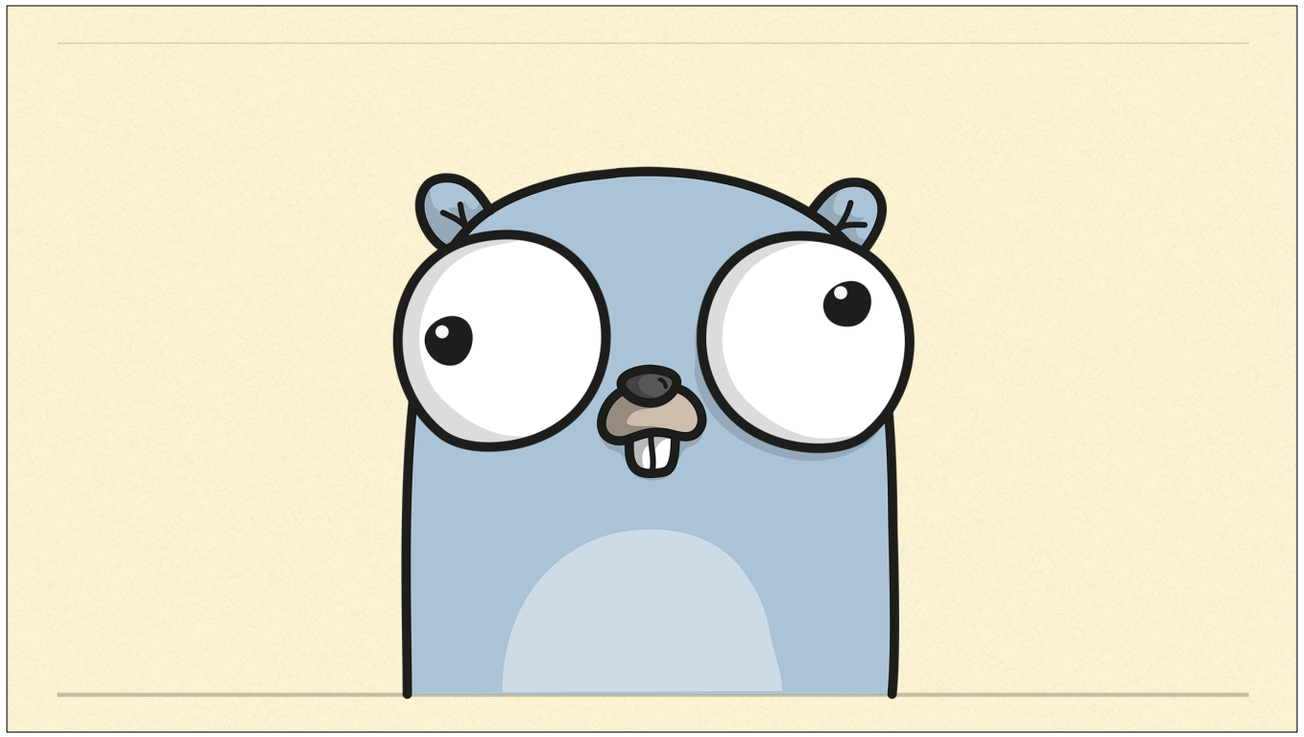
EVIL GO: HOW TO WRITE BAD CODE

Jon Bodner

Hi, my name is Jon Bodner, I'm a software engineer at Capital One, and after decades of programming in Java, I have been mostly working in Go for the past several years. And I suspect that this is not all that uncommon: how many people here used to work in a different language?



Now, working in Go is great, primarily because the code is so easy to follow. Without inheritance, manual memory management, function overloading, operator overloading, Go encourages a simple, straightforward programming style. Straightforward code is readable code and readable code is maintainable code. And sure, that's great for my company and co-workers, but what about me?

I started to wonder if all of this easy-to-understand code was in my best interest. Maybe I did want some unmaintainable code. After all, unmaintainable code is job security, right? How could I bring bad design to my Go programs? And then I thought, I bet there are other Go developers out there who want job security too! So I put together this talk to cover everything that you can do to make your Go code as hard to follow as possible.

HOW TO BE EVIL

- Poor Packaging
- Improper Interfaces
- Pass Pointers for Population
- Propagating Panics
- Side-Effect Set-up
- Complicated Configuration
- Frameworks For Functionality

We don't have time to cover all the ways you can make your code terrible, so let's just talk about a few. They are:

- Poor Packaging
- Improper Interfaces
- Pass Pointers for Population
- Propagating Panics
- Side-Effect Set-up
- Complicated Configuration
- Frameworks For Functionality

In order to properly do evil, you have to know how to do good. We'll go through these points, one at a time, look at how goody-goody programers would do things, and then see how to be evil.

POOR PACKAGING

Packages are a nice place to start, because they are so unassuming. How can organizing your code make a difference?

PACKAGE NAMES
DESCRIBE
CONTENTS

- fmt.PrintIn
- http.RegisterFunc
- json.Marshal

In Go, the name of the package is used as part of the name used to refer to the exported item, we write `fmt.Println` or `http.RegisterFunc`. Because the package name is so visible, good Go developers make sure that the package name describes what the exported items are. We shouldn't have packages named util because we don't want names like `util.JSONMarshal`, we want names like `json.Marshal`.

Good Go developers don't create a single dao or model package. For those who aren’t familiar with the term, a dao is a data access object, the layer of code that talks to your database.

If you have a single library for your company with all of your DAOs, that means that everyone shares a single dependency. The more of these single dependency chokepoints that are created, the more likely you'll end up with a cyclic dependency between packages, and Go doesn't allow that. You also might end up with a situation where a change to support one service requires you to upgrade all of your services, or else something will break. That’s called a distributed monolith and it’s incredibly difficult to do updates once you’ve built one.

USE BAD PACKAGE NAMES AND ORGANIZATION

- dao
- model
- util

So it's easy to be evil here. Organize your code badly and give your packages bad names. Break your code up into packages like model and util and dao and then when people end up with cyclic dependencies or distributed monoliths because they tried to use your code in a slightly new way, you get to sigh and tell them they are doing it wrong.

IMPROPER INTERFACES

Now that we have our packages all messed up, Let's move on to interfaces. Interfaces in Go are not like interfaces in other languages. The fact that you don't explicitly declare that a type implements an interface seems like a small detail at first, but it actually completely flips around the concept of interfaces. Let’s take a look why.

```java
public interface AccountDao {
    void CreateAccount(Account account);
    Account GetById(int id);
    List<Account> GetByName(name String);
    List<Account> GetByDateCreated(Instant date);
    void UpdateAccount(Account account);
    void TransferBetweenAccounts(int amt,
                                 int acct1, int acct2);
    voidTransferAccount(int account1, String name);
    void DeleteAccount(int id);
}

public class AccountDaoImpl implements AccountDao {
    //all of the implementations go here
}
```

In most languages with abstract types, the interface is defined before or alongside the implementation. You have to do this because sooner or later, you will need to swap in different implementations of an interface, if only for testing. If you don't create the interface ahead of time, you can't go back later and slip an interface in without breaking all of the code that uses the class, because they'll have to be re-written to refer to the interface instead of the concrete type.

For this reason, Java code often has giant service interfaces written with lots of methods. Classes that depend on these interfaces then use the methods they need and ignore the rest. Writing tests is possible, but you've added an extra layer of abstraction and you have to fall back on tools to generate implementations of all those methods that you don't care about.

```go
package account

type AccountCreator interface {
    CreateAccount(account Account)
}

func createNewAccount(ac AccountCreator, a Account) {
    ac.CreateAccount(a)
}
```

In Go, the implicit interfaces define what methods you need to use. It’s the using code that owns the interface, not the providing code. Even if you are using a type with tons of methods defined on it, you can specify an interface that only includes the methods that you need. And different code that uses different parts of the same type will define different interfaces that only cover the functionality that they need. Usually, these interfaces only have a couple of methods . This makes your code easier to understand, because not only does your method or function declaration define what data it needs, it also defines exactly what functionality it’s going to use. That’s why good Go developers follow the advice: "accept interfaces, return structs".

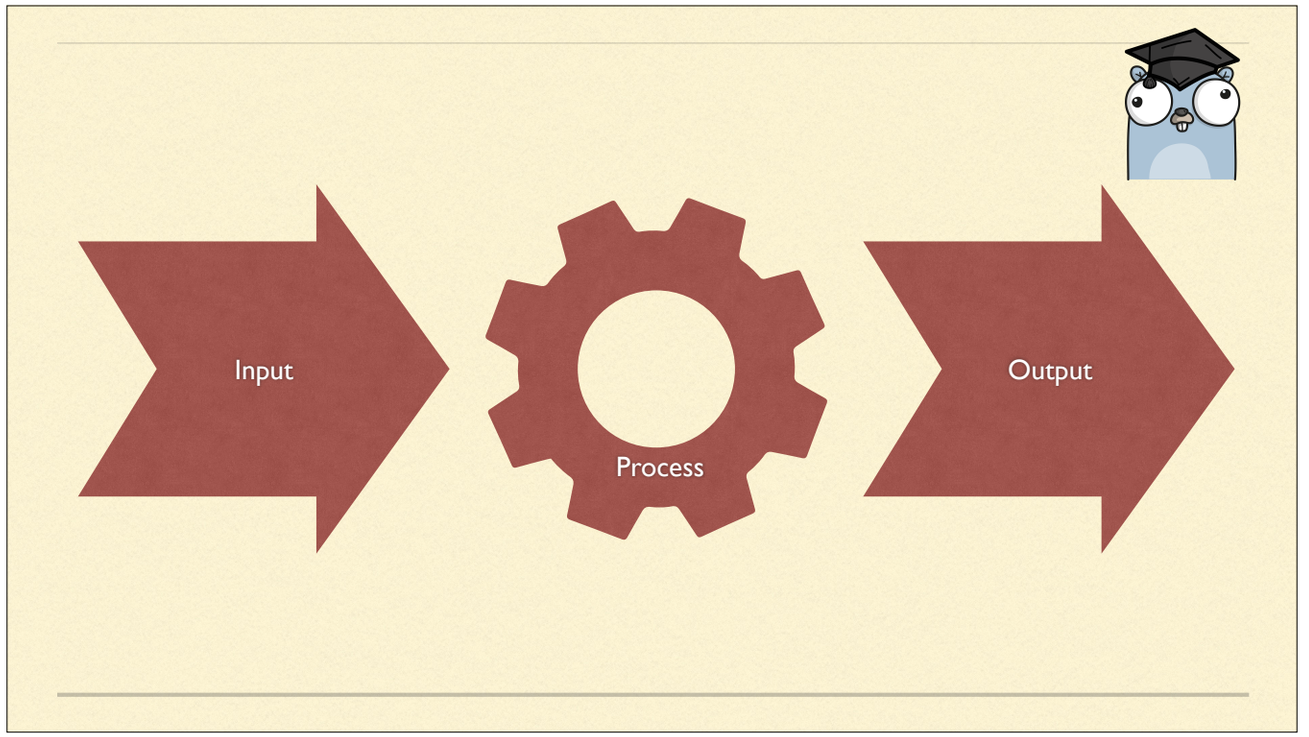
But, you know, just because it's a good practice doesn't mean that you _have_ to do it.

```go
package dao

import "github.com/evil-go/example/model"

type AccountDao interface {
    CreateAccount(account model.Account)
    GetById(id int) model.Account
    GetByName(string name) []model.Account
    GetByDateCreated(t time.Time) []model.Account
    UpdateAccount(account model.Account)
    TransferBetweenAccounts(amt int, acct1 int, acct2 int)
    TransferAccount(account1 int, name string)
    DeleteAccount(id int)
}
```

The best way to make your interfaces evil is to go back to what other languages do and define interfaces ahead of time, as part of the code that's called. Define really big interfaces, with lots of methods, that are shared by all of the clients of the service. That makes it unclear which methods are actually needed, which creates complication and complication is the friend of the evil programmer.

PASS POINTERS FOR POPULATION

On to our next bit of evil, pointer passing for population. Before we talk about what this means, we're going to get a little philosophical.



If you step back and think about it, every single program ever written does exactly the same thing: the program takes in data, it processes the data, and it sends the processed data someplace. And if you look at programs this way, the most important thing you can do is make sure that it is easy to understand how the data is being transformed. And that's why it's best to keep your data as immutable as possible, because data that doesn't change is data that's easy to track.

```
func main() {
    a := 1
    b := a
    b = 2
    fmt.Println(a, b) // prints 1 2

    c := &a
    *c = 3
    fmt.Println(a, b, *c) // prints 3 2 3
}
```

In Go, we have reference types and value types. Very quickly, the difference between the two is whether or not the variable refers to the data or to the data's location in memory. Pointers, slices, maps, channels, and functions are reference types, and everything else is a value type. If you assign a variable of a value type to another variable, it makes a copy of the value; a change to one variable doesn't change the other's value. Assigning a reference type variable to another reference type variable means they both share the same memory location, so if you change the data pointed to by one, you change the data pointed to by the other. And this is true for both local variables, and for parameters to functions.

```
type Foo struct {
    A int
    B string
}

func getA() int {
    return 20
}

func getB(i int) string {
    return fmt.Sprintf("%d",i*2)
}
```

Good Go developers want to make it easy to understand how data is gathered. They make sure to pass in value variables as often as possible. Go doesn't have way to mark the fields in a struct final, but if a function has value parameters, it can’t modify the values of its parameters. All it can do is return a value.

```
func main() {
    f := Foo{}
    f.A = getA()
    f.B = getB(f.A)
    //I know exactly what went into building f
    fmt.Println(f)
}
```

So if you populate a struct by calling functions that take value parameters, you can then composed the values into the struct and it is clear exactly where each value in the struct came from.

```go
type Foo struct {
    A int
    B string
}

func setA(f *Foo) {
    f.A = 20
}

//Secret dependency on f.A hanging out here!
func setB(f *Foo) {
    f.B = fmt.Sprintf("%d", f.A*2)
}
```

So how do we be evil? We do it by inverting this model. Rather than calling functions that return values that we compose together, we're going to pass a pointer to our struct into functions, and let them make changes to the struct. Since every function has the whole struct, the only way to know which fields are being modified is to look through all of the code. You can have invisible dependencies between the functions, too, with one function putting data in that a second function needs, but nothing in the code to indicate that you must call the first function first.

```
func main() {
    f := Foo{}
    setA(&f)
    setB(&f)
    //Who knows what setA and setB
    //are doing or depending on?
    fmt.Println(f)
}
```

If you build your data structures this way, you'll be sure that no one else will understand what your code is doing.

PROPAGATING PANICS

And now we're on to error handling.

```go
func (dus DBUserService) Load(id int) (User, error) {
    rows, err := dus.DB.Query("SELECT name FROM USERS WHERE ID = ?
    if err != nil {
        return User{}, err
    }
    if !rows.Next() {
        return User{}, fmt.Errorf("no user for id %d", id)
    }
    var name string
    err = rows.Scan(&name)
    if err != nil {
        return User{}, err
    }
    err = rows.Close()
    if err != nil {
        return User{}, err
    }
    return User{Id: id, Name: name}, nil
}
```

Maybe you're thinking that it's pretty evil to have programs that are roughly 75% error handling, and I wouldn't say you were entirely wrong. Go code has a lot of error handling, front and center. And sure, it would be nice if there was a way to make it a bit less in your face. But errors happen and how you handle errors is what separates the professionals from the amateurs. Bad error handling produces unstable programs that are hard to debug and hard to maintain. Sometimes being good means doing the hard work.

```java
public class Engine {
    private int curState;

    public void changeState(int newState) {
        if (newState < curState) {
            throw new IllegalStateException(
                                    "Can't go backwards!");
        }
        curState = newState;
    }
}
```

A lot of languages, C++, Python, Ruby, Java, they use exceptions to handle errors. If something goes wrong, just throw or raise an exception, and someone, somewhere will take care of it. Probably. If they know that it was even possible for the exception to be thrown, because, with the (no pun intended) exception of Java's checked exceptions, there's nothing in the function or method signature to tell you that an exception might happen. So how do you know what errors to worry about? You can either read through all of the source code of all the libraries that your code calls, and all the libraries that that code calls, and so on, or you can trust the documentation. Raise your hands if you trust the documentation. Yeah, thought so.

```
func PanicIfErr(err error) {
    if err != nil {
        panic(err)
    }
}
```

So how do we bring this brand of evil to Go? By abusing the panic and recover keywords. Panic is meant for situations like "disk disappeared" or "network card exploded". It's not for things like "someone passed a string instead of an int". But it could be. Unfortunately, other, less enlightened people will be returning errors from their code, so here’s a little helper function, PanicIfErr. Use that to turn other people’s errors into panics.

```
func (dus DBUserService) LoadEvil(id int) User {
    rows, err := dus.DB.Query(
                "SELECT name FROM USERS WHERE ID = ?", id)
    PanicIfErr(err)
    if !rows.Next() {
        panic(fmt.Sprintf("no user for id %d", id))
    }
    var name string
    PanicIfErr(rows.Scan(&name))
    PanicIfErr(rows.Close())
    return User{Id: id, Name: name}
}
```

And once you have PanicIfErr, you can wrap other people's errors, shrink your code down, no more ugly error handling. Anything that you would have made an error is now a panic. It's so productive!

```go
func PanicMiddleware(h http.Handler) http.Handler {
    return http.HandlerFunc(
        func(rw http.ResponseWriter, req *http.Request){
            defer func() {
                if r := recover(); r != nil {
                  fmt.Println("Yeah, something happened.")
                }
            }()
            h.ServeHTTP(rw, req)
        }
    )
}
```

You can put a recover somewhere near the top, maybe in your own middleware, and say that not only are you are handling the errors, you have made everyone’s code cleaner too. Doing evil by looking like you are doing good is the best kind of evil.

SIDE-EFFECT SET-UP

Next we're on to configure by side effect.

```
package main

import (
    “github.com/evil-go/example/account"
)

func main() {
    a := StubAccountService{}
    money := a.GetBalance(12345)
}
```

Remember, a Good Go developer wants to understand how data flows through their program. Well, the best way to do that is to know what the data is flowing through, by explicitly configuring the dependencies in an application. Even things that meet the same interface may have very different behaviors to meet that interface, like the difference between code that stores data in memory versus code that calls a database to do the same work.

```go
package account

type Account struct{
    Id int
    UserId int
}
```

```go
func init() {
    fmt.Println("I run magically!")
}
```

```go
func init() {
    fmt.Println(
       "I also run magically, and I am also named init()")
}
```

Lots of languages have way to magically run code without explicitly calling it and Go isn't immune. Who in the audience has heard of init functions? If you create a function called `init` with no parameters, it automatically runs whenever a package is loaded, no need to invoke it explicitly. And, just to make it more fun, you can actually have multiple functions named init in a single file, or across multiple files in a single package.

```
package main

import _ "github.com/lib/pq"

func main() {
    db, err := sql.Open(
        "postgres",
        "postgres://jon@localhost/evil?sslmode=disable")
}
```

And who here has heard of blank imports? You can write a line like `import _ "github.com/lib/pq` and what this does is import package github.com/lib/pq, run its init methods, but not give you access to any of the identifiers in the package. For some Go libraries, like database drivers and image formats, you have to load them by including a blank import for the package somewhere in your application, just to trigger the init function in the package so it can register some code.

Now, this is obviously a bad idea. You have code that runs magically, completely out the control of the developer. And use of init() is pretty discouraged right now. It's obscure, it's obfuscated, and it's easy to hide in a library.

```go
package account

import (
    "fmt"
    "github.com/evil-go/example/registry"
)

type StubAccountService struct {}

func (a StubAccountService) GetBalance(accountId int) int {
    return 1000000
}

func init() {
    registry.Register("account", StubAccountService{})
}
```

In other words, it's perfect for our evil purposes. Rather than having explicit configuration or registration of items in your packages, you can use init functions and blank imports to set up your application's state. In this example, we’re making account available to the rest of the application via a registry and the account package puts itself into the registry using an init function.

```go
package main

import (
    _ "github.com/evil-go/example/account"
    "github.com/evil-go/example/registry"
)

type Balancer interface {
    GetBalance(int) int
}

func main() {
    a := registry.Get("account").(Balancer)
    money := a.GetBalance(12345)
}
```

Then if you want to use an account, you put a blank import somewhere in your program. It doesn't have to be main, it doesn't have to be related code, it just has to be _somewhere_. It's magic! Just put some inits into your libraries, and watch people scratch their heads, wondering how dependencies were set up and how to change them. Say it with me: job security.

COMPLICATED CONFIGURATION

So let's talk some more about configuring your application.

```
func main() {
    b, err := ioutil.ReadFile("account.json")
    if err != nil {
        fmt.Errorf("error reading config file: %v", err)
        os.Exit(1)
    }
    m := map[string]interface{}{}
    json.Unmarshal(b, &m)
    prefix := m["account.prefix"].(string)
    maker := account.NewMaker(prefix)
}
```

If you're being good, you want to isolate the configuration from the rest of the program. You're using the main for your program as a way to to capture properties from the environment and convert them into the values that are needed by the components that are explicitly wired together. Your components don't know anything about property files or how those properties are named.

```go
type Maker struct {
    prefix string
}

func (m Maker) NewAccount(name string) Account {
    return Account{Name: name, Id: m.prefix + "-12345"}
}

func NewMaker(prefix string) Maker {
    return Maker{prefix: prefix}
}
```

For simple components, you have public properties to set and if you want to be fancy, you can have a factory function that takes in the configuration information and returns a properly configured component.

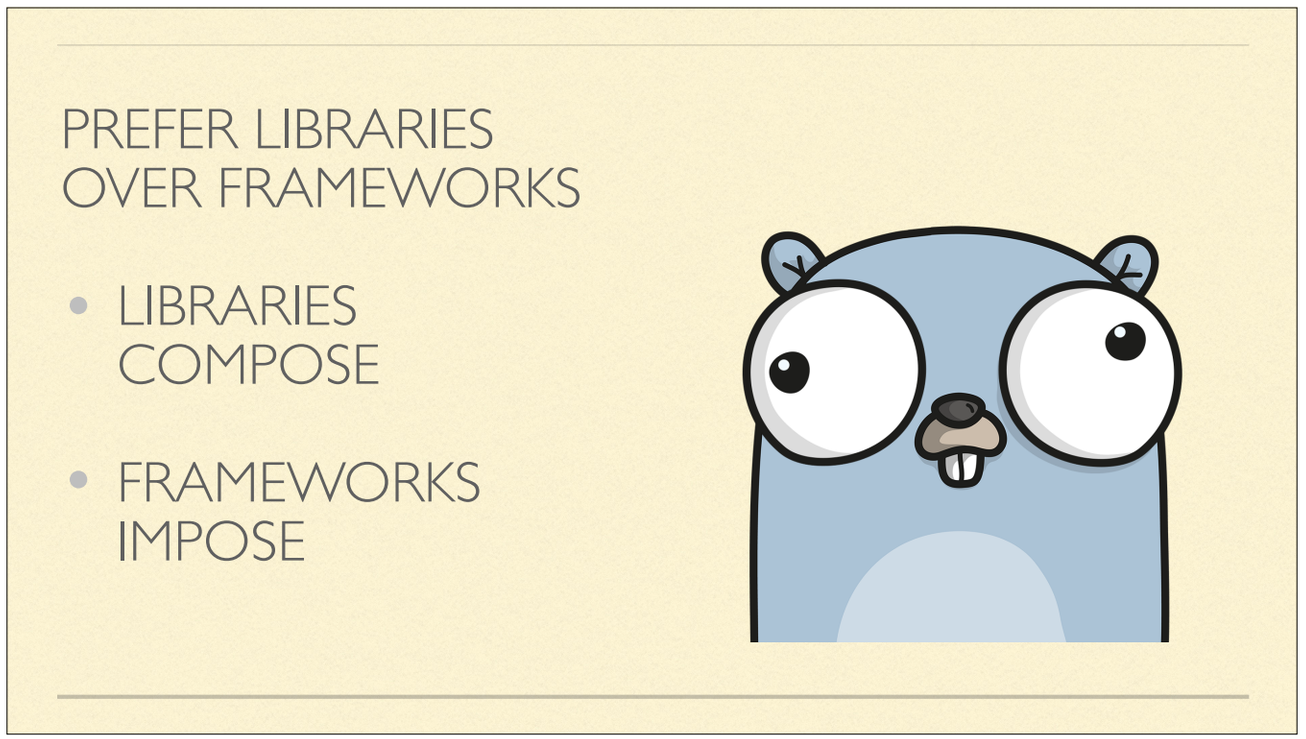
```go
func (m maker) NewAccount(name string) Account {
    return Account{Name: name, Id: m.prefix + "-12345"}
}

var Maker maker

func init() {
    b, _ := ioutil.ReadFile("account.json")
    m := map[string]interface{}{}
    json.Unmarshal(b, &m)
    Maker.prefix = m["account.prefix"].(string)
}
```

But evil programmers know that it's best to sprinkle configuration information throughout the entire program. Don't have a function that lists the values you need, just take in a map of string to string and convert them yourself. If that seems too obviously hostile, use an init function to load a property file from inside of your package and set up the values yourself. It seems like you have made life easier for other people, but you know better.

With an init function, you can define new properties deep within the code and no one will ever find them until they get to production and everything crashes because something is missing from one of the dozen different properties files that are required in order to launch correctly. If you want extra evil powers, you can offer to set up a wiki to track all of the properties across all of the libraries and "forget" to include new properties periodically. As the keeper of the properties, you become the only person who can get software to run.

FRAMEWORKS FOR FUNCTIONALITY

Finally, we come to frameworks vs. libraries. The difference is subtle.



It's not just a size thing; you can have large libraries and small frameworks. A framework calls your code, while you call a library's code. Frameworks require you to write your code in a certain way, whether it's naming your methods just so, or making sure they meet certain interfaces or making you register your code with the framework. Frameworks dump their requirements all over your code. In general, frameworks own you.

Good Go encourages libraries because libraries are composable. While, sure, every library expects data to be passed in in a certain format, you can write a little bit of glue code to massage the output of one library into the input for another.

FRAMEWORK ALL THE THINGS

WRITE AN IN-HOUSE FRAMEWORK FOR MAXIMUM EVIL

With frameworks, it's hard to get them to play together nicely, because each framework wants complete control over the lifecycle of the code that runs inside of it. Oftentimes, the only way to get frameworks to work together is for the authors of the frameworks to get together and put in explicit support for each other. And the best way to use the evil of frameworks to gain lasting power: write a custom framework that is only used in-house.

BIG BALL OF EVIL

- Framework with:
  - Lots of Configuration Files
  - Populates Fields of Structs using Pointers
  - Defines Interfaces For Published Types
  - Magically Runs Code
  - Lots of Panics

So what does it look like if we put all of this evil together? We'd have a framework that used lots of configuration files, populated fields of structs using pointers, defined interfaces to describe the types that are being published, relied on magically-run code, and panicked whenever there was a problem.

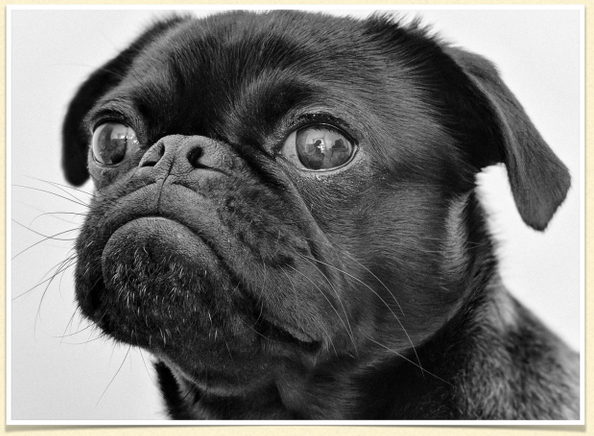
FALL

- https://github.com/evil-go/fall

If you go to github.com/evil-go, you will see Fall, a dependency injection framework designed to bring you all of the bad things you could possibly want.

OUTBOY

- https://github.com/evil-go/outboy



(PHOTO BY CHARLES DELUVIO ON UNSPLASH)

I've paired it with a tiny web framework, outboy, that follows the same principles. And, yes, they are real.

Now, you might be saying, how bad could it really be? Well, let's take a look. We can't go through a big program today, but let’s walk through a simple Go program that exposes a single http endpoint, and then rewrite the program using Fall and outboy.

```
type Dao struct {
  DefaultMessage string
  BobMessage     string
  JuliaMessage   string
}
```

```
func (sdi Dao) GreetingForName(name string) (string, error) {
  switch name {
  case "Bob":
    return sdi.BobMessage, nil
  case "Julia":
    return sdi.JuliaMessage, nil
  default:
    return sdi.DefaultMessage, nil
  }
}
```

Our simple program is based around a single package called greet, that contains all of our basic functionality to implement our single endpoint. Because this is a sample, we have a dao that works entirely in memory, with three fields to hold the different values we will return. There's also a method that fakes our database call and returns the proper greeting, depending on the input to the method.

```go
type Response struct {
  Message string
}
```

```go
type GreetingFinder interface {
  GreetingForName(name string) (string, error)
}
```

```go
type Service struct {
  GreetingFinder GreetingFinder
}
```

Next comes our business logic, and in order to implement our business logic, we define a struct to hold the output from the business logic, a GreetingFinder interface to describe what the business logic is looking for in a data lookup layer, and the struct to hold the business logic itself, with a field for the GreetingFinder.

```go
func (ssi Service) Greeting(name string)(Response, error) {
  msg, err := ssi.GreetingFinder.GreetingForName(name)
  if err != nil {
    return Response{}, err
  }
  return Response{Message: msg}, nil
}
```

The actual logic is simple, it just calls the GreetingFinder, and handles any errors that might happen.

```
type Greeter interface {
  Greeting(name string) (Response, error)
}
```

```
type Controller struct {
  Greeter Greeter
}
```

Next comes the web layer, and for that part, we define a Greeter interface that provides all the business logic we need, and a struct to contain our http handler that is configured with a Greeter

```
func (mc Controller) ServeHTTP(rw http.ResponseWriter,
                               req *http.Request) {
  result, err := mc.Greeter.Greeting(
                               req.URL.Query().Get("name"))
  if err != nil {
    rw.WriteHeader(http.StatusInternalServerError)
    rw.Write([]byte(err.Error()))
    return
  }
  rw.Write([]byte(result.Message))
}
```

Then we have the method to implement the http.Handler interface, which takes apart the http request, calls the greeter, handles any errors, and returns back the results. Pretty simple stuff. That’s the end of the greet package, let’s take a look at how we build our main.

```go
type Config struct {
  DefaultMessage string
  BobMessage     string
  JuliaMessage   string
  Path           string
}
```

```go
func main() {
  c, err := loadProperties()
  if err != nil {
    fmt.Println(err)
    os.Exit(1)
  }
```

In the main package, we've got a struct called Config that defines the properties that we need in order to run. And we have a main function that does three things. First, it load the properties from a properties file and puts them into an instance of config. We're running short on time, so we'll skip over the implementation of load properties, but it's a small library that you can find online. If config loading fails, we report an error and exit.

```
  dao := greet.Dao{
    DefaultMessage: c.DefaultMessage,
    BobMessage:     c.BobMessage,
    JuliaMessage:   c.JuliaMessage,
  }
  svc := greet.Service{Dao: dao}
  controller := greet.Controller{Service: svc}
```

```
  err = server.Start(server.Endpoint{
                       c.Path, http.MethodGet, controller})
  if err != nil {
    fmt.Println(err)
    os.Exit(1)
  }
}
```

Next, the main function wires up the components in the greet package, explicitly assigning them values from the config and setting up how one component depends on another. Finally, it calls a small server library, passes in an endpoint description of the path to be handled, the http verb that should be listened for, and the http.Handler to process the request, and the server starts up. This example is pretty short, but it shows how good Go is written; some things are a little verbose, but it’s clear what’s going on. We glue together small libraries that are explicitly configured to work together. Nothing is hidden; anyone could pick up this code, understand how the parts all fit together, and swap in new parts when needed.

```
package dao

import (
    "github.com/evil-go/fall"
    "os"
    "fmt"
)

type GreetDao interface {
    GreetingForName(name string) string
}
```

Now we’re going to look at the Fall + outboy version. First of all, we’re going to break apart our greet package into multiple packages, each one containing one layer in the application. Here’s the dao package. We import fall, our dependency injection framework, and because we are being evil and defining interfaces on the wrong side of the relationship, we define an interface called GreetDao. Notice that we’ve removed any reference to an error; if something fails, we’re just going to panic. So far, we’ve got bad packaging, bad interfaces, and bad errors, so we’re off to a great start.

```
type greetDaoImpl struct {
    DefaultMessage string `value:"message.default"`
    BobMessage string `value:"message.bob"`
    JuliaMessage string `value:"message.julia"`
}

func init() {
    fall.RegisterPropertiesFile(“dao.properties")
    fall.Register(&greetDaoImpl{})
}
```

Next we see our struct from before, slightly renamed, but we’ve added struct tags to our fields. They have a key value, and that key tells fall to assign a value that’s registered with fall to the field. The method is mostly the same, so we can skip it, but we also now have an init function for our package, And that’s where we really start piling on the evil. In the package’s init function, we call fall twice; once to register a properties file that supplies values for those struct tags, and once to register a pointer to an instance of the struct, so that fall can populate those fields for us and make our sample dao available for other code to use. Speaking of which, let’s look at the service layer for our code.

```
package service

import (
    "github.com/evil-go/sample/dao"
    "github.com/evil-go/sample/model"
    "github.com/evil-go/fall"
    "fmt"
)

type GreetService interface {
    Greeting(string) model.Response
}
```

So in the service layer, we import our dao layer, because we need to access the interface defined there, and we also import a model package that we haven’t seen yet, but we’ll store all of our data types there. And of course we import fall, because like all good frameworks, it forces itself everywhere. We also define an interface for the service to expose to the web layer, with again, no error returned.

```
type greetServiceImpl struct {
    Dao dao.GreetDao `wire:""`
}
```

```
func (ssi greetServiceImpl) Greeting(name string)
model.GreetResponse {
    return model.GreetResponse{
        Message: ssi.Dao.GreetingForName(name)
    }
}
```

```
func init() {
    fall.Register(&greetServiceImpl{})
}
```

And the implementation of our service, now has a struct tag with wire. What does that mean? Well, any field tagged with wire will have its dependency automatically assigned when the struct is registered with fall. What is going to be assigned to this field? In our tiny sample, it's clear, but in a bigger program, who knows? All that we know is that something, somewhere is going to implement that GreetDao interface. Maybe it talks to a database, maybe it's test code. Next we have our method for our service, which has been modified slightly to get the GreetResponse struct from the model package and which removes any error handling. And finally, we have our init, that registers an instance of our service with fall. Now let's take a look at that model package:

```
package model

type GreetResponse struct {
    Message string
}
```

There's not much to see here, just the model is separated from the code that creates it, for no reason other than layering.

Then we have our web front end in the web package:

```
package web

import (
    "net/http"
    "github.com/evil-go/outboy"
    "github.com/evil-go/fall"
    "github.com/evil-go/sample/service"
    "os"
    "fmt"
)

type GreetController struct {
    Service service.GreetService `wire:""`
    Path string `value:"controller.path.hello"`
}
```

In our web package, we import both fall and outboy, and we import the service package that our web package depends on. Because frameworks only play nice together when they integrate behind the scenes, Fall has some special-case code to make sure that it and outboy work together. We also modify a struct to be the controller for our web app, and it has two fields, one which is wired by fall with an implementation of that GreetService interface and the other which is the path for our single web endpoint. It’s assigned the value of a property that’s loaded from…somewhere?

```go
func (mc GreetController) GetHello(rw http.ResponseWriter,
                                   req *http.Request) {
    result := mc.Service.Greeting(
                               req.URL.Query().Get("name"))
    rw.Write([]byte(result.Message))
}
```

```go
func (mc GreetController) Init() {
    outboy.Register(mc, map[string]string{
        "GetHello": mc.Path,
    })
}
```

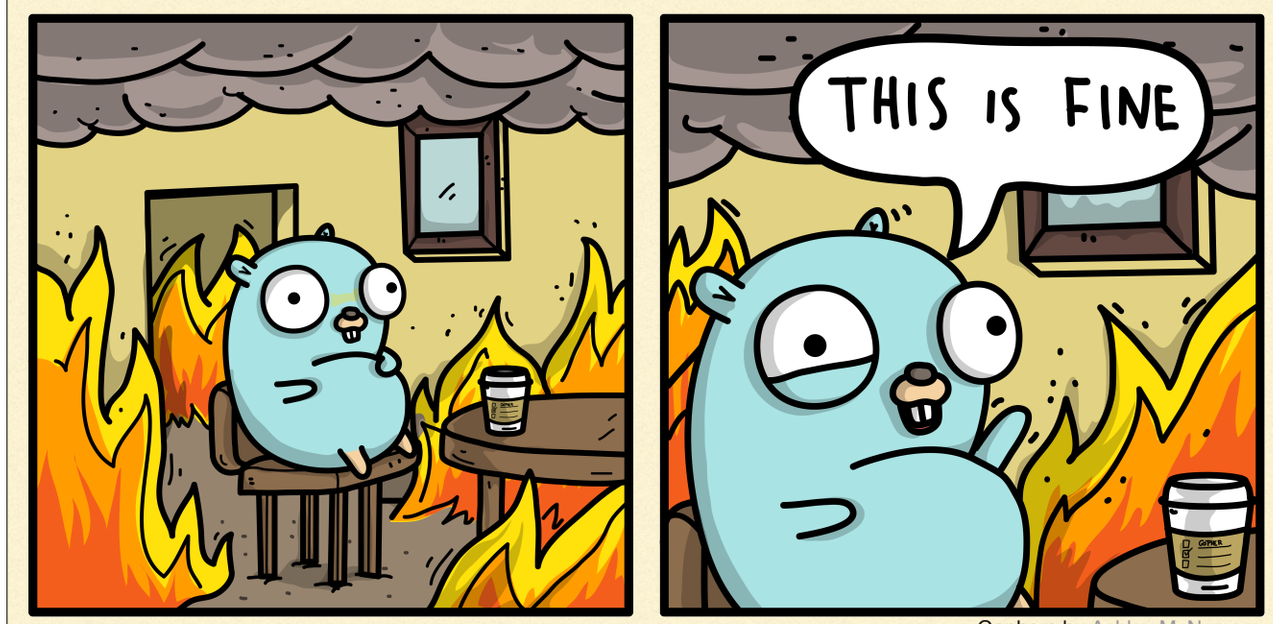
```go
func init() {
    fall.RegisterPropertiesFile("web.properties")
    fall.Register(&GreetController{})
}
```

Here's the rest of our web tier. Our http Handler is now renamed to GetHello and is error handling free. We also have an Init, capital I, method. That's a magic method that's invoked on structs registered with fall after all of their fields are populated. In capital-I init, we call outboy to register our Controller and its endpoint at the path that was assigned by fall. You might be wondering, we've got the controller, we've got the path, but where's the HTTP verb? In outboy, you look at the method's name and use that to figure out what verb a method responds to. Since our method is GetHello, it responds to GET requests. If you don't know the rules, you can't understand what kinds of requests it responds to. Evil, right?

Finally we call the init function and we register a property file with fall to get that path and we register our controller with fall as well. The only thing left to show is how we kick off a program:

```
package main

import (
    "github.com/evil-go/fall"
    _ "github.com/evil-go/outboy"
    _ "github.com/evil-go/sample/web"
)

func main() {
    fall.Start()
}
```

In main, we've got a couple of blank imports to register outboy and our web tier, and then we call fall.Start() and that kicks off the whole application. And there we go, a complete program written using all of our evil tools.



Gophers by Ashley McNamara

So, this is terrible. The magic is obscuring how the parts of your program fit together, making it far harder to understand what’s going on. And, yet, you have to admit, there's something...seductive...about writing code with Fall and Outboy. For a tiny program, you could even argue that it’s an improvement. Look at how easy it is to specify configuration information! I can wire up dependencies with almost no code! I registered a handler for a method, just by using its name! And without any error handling, everything looks so clean! And that's the way evil works, at first glance, it's really appealing. But as your program changes and grows, all of that magic starts getting in the way, making it harder to figure out what’s going on. It’s only when you are fully in the grip of evil do you look around and see that you're trapped.

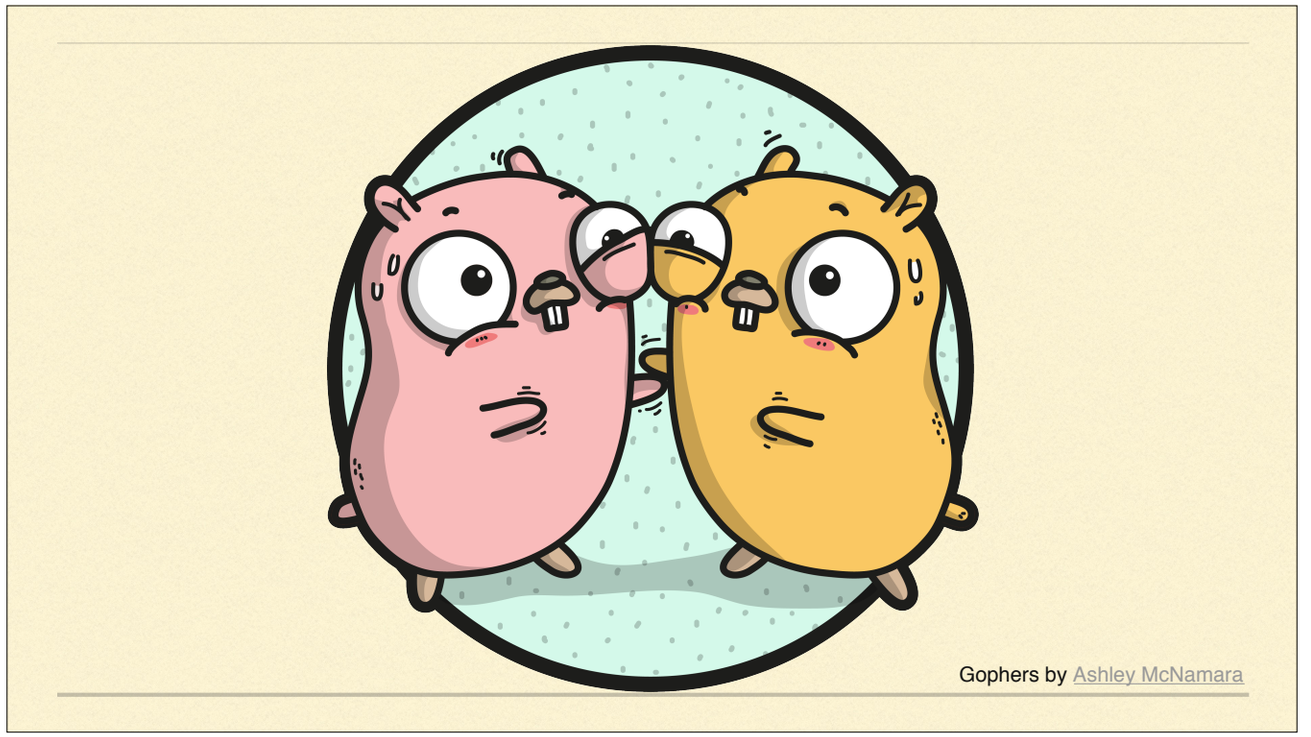
For the Java developers in the audience, this all might sound slightly familiar. If you aren't a Java dev, take a look around, find the people who are shuddering, and just whisper the word

"Spring".

> “The enterprise is coming.”
>
> *–Brian Ketelsen*

For everyone else, welcome to the nightmare. Because, as Brian Ketelsen has said, "The Enterprise is Coming". And what that means is that

Java developers are coming, and unfortunately, they have been trained in decades of bad practices. We can mock them, but there's a reason for those bad practices. Some are a result of language choices, and others are just the end result of decisions that made sense in context 20 years ago, but no longer do now.



Don't get me wrong. We should embrace the enterprise. Go was designed for programming in the large, for projects that span hundreds of developers and dozens of teams. But in order for Go to do that, it needs to be used the way that it works best. We can choose to be evil, or we can choose to be good. If we choose evil, we can encourage all the bad habits that enterprise Java developers will bring with them. Or, we can choose good. Part of our job as Go developers is to mentor these new Gophers, and understand the reasons behind our best practices well enough to explain why they should be followed.

The only downside of doing good is that you'll have to find another way to express your inner evil.

Photo by Thought Catalog on Unsplash

Can I interest anyone in bitcoin mining?

ANY QUESTIONS?

Thanks for your time. Do you have any questions?