Three reasons you should not use Martini

TL;DR: Martini makes Go magic, but the whole point of Go is to be simple and not magic. It's the wrong direction for the language and ecosystem, so try something like muxchain instead.

> *Sorry codegangsta; I really tried to write this post without calling out Martini, but the point just wasn't coming across.*

Martini has gained a tremendous amount of attention in a short time. I think it might have even gotten a few people to give Go a try. I think you shouldn't use it and I have several reasons.

1. Martini defeats the Go type system

Martini uses dependency injection to determine what your handlers are and how to pass requests to them and get responses back from them. Martini allows you to pass anything to its dependency injector and call it a handler.

Take a look at this code:

```
func main() {
 m := martini.Classic()
 m.Get("/bad", func() {
   fmt.Println("Did anything happen?")
 })
 m.Run()
}
```

We're going to revisit this example later, but this compiles, runs, and does *almost* what you'd expect. The gotcha, though, is that you would never want to do this, and appropriately, `net/http` would never allow it. That misstep is providing a handler that has no way of writing a response.

Another example:

```
func main() {
  m := martini.Classic()
  m.Get("/myHandler", myHandler)
  m.Run()
}

var myHandler = "print this"
```

Does this work? Answer: sort of. It compiles, which should indicate it works. However, as soon as you run it, it will panic. Since the typical use case will be to set up handlers during initialization, this might not become a significant pain point for actually specifying handlers, but this is a real problem for the dependency injection package that's part of Martini. This is abusing reflection so that you can defeat the compile-time type system and if you use it, it's only a matter of time before it bites you. Moreover, the type system is a huge reason for choosing Go, so if you don't want it, maybe try Python with Twisted.

2. Martini defeats one of Go's best patterns: uniformity and discoverability through streaming data

Say you're new to Go. You have some images you'd like to serve on the web. This perfectly valid code seems like a reasonable, if naive, approach:

```
func main() {
  m := martini.Classic()
  m.Get("/kitten", func() interface{} {
    f, _ := os.Open("kitten.jpg")
    return f
  })
  m.Run()
}
```

What do you get in the browser? `<*os.File Value>` Wrong. So we try again.

```
func main() {
  m := martini.Classic()
  m.Get("/kitten", func() interface{} {
    f, _ := os.Open("kitten.jpg")
    defer f.Close()
    i, _, _ := image.Decode(f)
    return i
  })
  m.Run()
}
```

Now we get: `<invalid Value>`. What? You can dig into what Martini is doing with `reflect` to fully understand why this happens, but it suffices to say that this happens if you return an `interface` type from a handler to Martini. It is conceivable that someone would write a handler returning the empty interface, as I have done here, and sometimes return a correct type to Martini, but in some rare case, return something invalid. That sounds like a potential production issue, and makes Martini a risk.

So we try again and finally we got it.

```
func main() {
  m := martini.Classic()
  m.Get("/kitten", func() interface{} {
    f, _ := os.Open("kitten.jpg")
    defer f.Close()
    b, _ := ioutil.ReadAll(f)
    return b
  })
  m.Run()
}
```

This works. We finally got the image. Careful readers will notice out what's terribly wrong here and why this anti-pattern should be avoided. All of the image is read into memory before being returned. If we took this approach with say, a large movie file, well, we might actually just run out of memory on the first request.

Others have written about Go's inheritance of the Unix philosophy, but Martini makes it deceptively easy to throw that baby out with the bathwater. Yes, yes, you can just pass an `http.Handler` to Martini, but if you don't always do that, we still lose the uniformity of our handlers (see the above point on types) and the discoverability of `io.Reader` and `io.Writer` beauty. To show you what I mean, here's the handler I would have written:

```
func main() {
m := muxchainutil.NewMethodMux()
m.("GET /kitten", func(w http.ResponseWriter, req *
http.Request) {
    f, _ := os.Open("kitten.jpg")
```

```
    defer f.Close()
    io.Copy(w, f)
})
http.ListenAndServe(":3000", m)
```

This way, the `io` does all the work for me and uses only a small amount of memory as a buffer between the image and the network. Because `io.Reader` and `io.Writer` are used throughout the standard library and beyond, you end up being able seamlessly tie together things like images, encryption, HTML templates, and HTTP. It's wonderful and it's probably one of the biggest unsung heros of Go. However, HTTP is probably the most commonly used case. It's how I discovered how awesome this is. Let's not hide it, and let's definitely not break it.

3. Martini is broken

```
package main

import (
 "fmt"

 "github.com/go-martini/martini"
)

func main() {
 m := martini.Classic()
 m.Get("/", func() string {
   return "hello world"
 })
 m.Get("/bad", func() {
   fmt.Println("Did anything happen?")
 })
```

```
 m.Run()
}
```

The `/bad` route prints to `stdout`, but logs a status code of 0 (no response) while it actually returns a 200. What should it do here? That's unclear, but the 200 response is default `net/http` functionality and the 0 response is inconsistent.
Ok, given this code:

```
import (
 "io"
 "net/http"

 "github.com/go-martini/martini"
)

func main() {
 m := martini.Classic()
 m.Get("/+/hello.world", func(w http.ResponseWriter
, req *http.Request) {
 io.WriteString(w, req.URL.Path)
 })
 m.Run()
}
```

what would you expect to happen if you browse to `/+/hello.world`? Yes, that's right, a 404. Ok, now how about `//hello.world`? That one works. Have you figured it out? Martini is interpreting this as a regular expression. This is an inherent flaw with Martini as a Go package. Because you're passing Martini this string and it might be a regular expression, but it might not, and some regular expressions are valid URL paths, there's potential for conflict.

Conclusion

This post isn't really about Martini. I singled out that package because I keep seeing it everywhere and have to keep asking myself why. I understand that it provides a great deal of convenience, but it does so with magic that comes at an even greater cost. One could see a future where many packages repeat these mistakes, pollute the ecosystem, and lead to difficulty in finding packages that don't compound bugs like I have pointed out. This is not a good direction for us to take with Go. The wonderful thing, though, is that you don't need it. I wanted to prove this to myself, so I wrote the muxchain package. Between that and muxchainutil, I have provided nearly every feature Martini provides, but with no magic. Everything is literally `net/http`, and there's no reflection. Give it a try and tell me what you think. It's new and might need some work. Most of all, I'm sure the naming could use some improvement. (Codegangsta: credit where credit's due, that's one thing you totally got right with Martini, so if you have any suggestions, please let me know.)

Categorized in: Code

Posted on May 16, 2014 by stephen