Logo and gophers were designed by @marcusolsson and animated by me

Why?

We make software overly complex

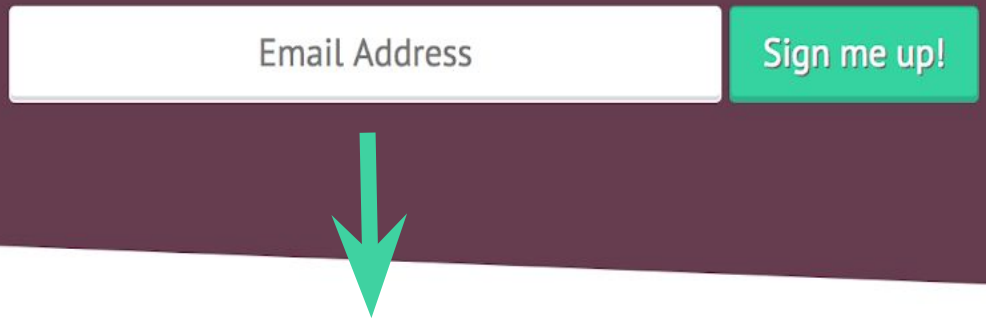
“Your web app should...

- Use a JSON API w/ microservice design
- Use Docker + Kubernetes to orchestrate deploys
- Avoid frameworks and ORMs
- … and the list goes on

Content Management System

```
app.Exercise{
  Number:      1,
  Title:       "Quiz Game",
  Link:        "/quiz",
  Description: "Create a program to run timed quizzes via ...",
  Topics:      []string{"strings", "channels", "goroutines", "flags"},
  Duration:    36*time.Minute + 52*time.Second,
  Videos: []app.Video{
    app.Video{Name: "Overview", VimeoID: "238136894"},
  },
  Solutions: []app.Solution{
    app.Solution{Name: "Part 1", Branch: "solution-p1"},
  },
},
```

Non-traditional Authentication



Hi there,

Thanks so much for checking out Gophercises! To access the course simply click following link:

https://gophercises.com/login?token=some-token-here

The link will take care of logging you in to your account - no password necessary. If you ever lose this email you can enter your email address on the Gophercises.com page and I'll send you another copy

Deploying

```
# 1. Build the app
$ mage prod

# 2. Upload the binary to the production server
$ rsync -azP prod root@gophercises.com:/path/to/app/prod

# 3. Stop the service on the server
$ ssh root@gophercises.com "sudo service gophercises.com stop"

# 4. Reseed the database with exercise data
$ ssh root@gophercises.com "/path/to/app/prod seed --db /path/to/db"

# 5. Restart the application on the server
$ ssh root@gophercises.com "sudo service gophercises.com restart"
```

Packr for serving assets

```
// Assets are compiled into the binary with gobuffalo/packr
images := packr.NewBox("../assets/img")


// Assets can be accessed via the Bytes method
imageBytes := images.Bytes(imagePath)


// Packr boxes can also be used as an http.FileSystem and then
// served via the http.FileServer handler
mux.Handle("/img/",
  http.StripPrefix("/img", http.FileServer(images)))
```

Mage, Cobra, and BoltDB

```
# Mage is used for multiple build targets
$ mage
Targets:
  dev    Builds the development binary that reads assets from disk
  prod   Builds the production binary with embedded assets


# Cobra allows me to build one binary with many subcommands
$ mage dev
$ ./app server --db /path/to/db # starts the server
$ ./app seed --db /path/to/db   # seeds the database
```

My build and deploy process focuses on my needs

Rendering HTML pages

Server side rendering via the `html/template` package

Code organization:

- Smaller, component-like templates
- Decorator pattern
- Service objects

Smaller, component-like templates

```
{{define "exerciseWidget"}}
<div class="panel widget widget-exercise">
  <!-- ... some code omitted for brevity -->
  <div class="col-xs-12">
    <p class="mb0">Exercise {{.Number}}</p>
    <h4 class="m0">{{.Title}}</h4>
  </div>
  <!-- ... -->
  <div class="col-xs-4 text-right text-top">
    <p class="mb0">Length</p>
    <p class="m0 h4 length">{{.Duration}}</p>
  </div>
  <!-- ... -->
</div>
{{end}}
```

Decorator pattern

```
package app

type Exercise struct {
  Duration  time.Duration
  // + other fields
}
```

Decorator pattern

```
package app

type Exercise st
  Duration  time
  // + other fie
}
```

```
package html

type Exercise struct {
  app.Exercise
}
```

Decorator pattern

```
package app

type Exercise st
  Duration  time
  // + other fie
}
```

```
package html

type Exercise struct {
  app.Exercise
}

func (e Exercise) Duration() string {
  if e.Exercise.Duration < 0 {
    return "TBD"
  }
  // returns a string like “1:22:03”
  return fmt.Sprintf("%d:%02d:%02d", e.Hours(),
    e.Minutes(), e.Seconds())
}
```

html/template + decorators

```
{{define "exerciseWidget"}}
<div class="panel widget widget-exercise">
  <!-- ... some code omitted for brevity -->
  <div class="col-xs-12">
    <p class="mb0">Exercise {{.Number}}</p>
    <h4 class="m0">{{.Title}}</h4>
  </div>
  <!-- ... -->
  <div class="col-xs-4 text-right text-top">
    <p class="mb0">Length</p>
    <p class="m0 h4 length">{{.Duration}}</p>
  </div>
  <!-- ... -->
</div>
{{end}}
```

Service objects

```
type UserCreator struct {
  userService *db.UserService
  mgClient    *mailgun.Client
}

func (uc *UserCreator) Create(email string) (string, error) {
  token, err := uc.userService.Create(email)
  if err != nil {
    return "", err
  }
  err = uc.mgClient.Welcome(email, token)
  return token, err
}
```

HTTP Handlers are simplified

```
type Users struct {
  userCreator *service.UserCreator
}

func (u Users) Signup(w http.ResponseWriter, r *http.Request) {
  // 1. Parsing incoming data
  email := r.PostFormValue("email")

  // 2. Actually doing the “signup” work
  _, err := u.userCreator.Create(email)

  // 3. Rendering results
  if err != nil { http.Error(...) }
  http.Redirect(...)
}
```

Gophercises has no tests

I’m a big fan of tests - my next course is on testing in Go!

Gophercises is simple and rarely changes

I wanted to understand the problem better before writing tests

Key takeaway

You can build simpler software in Go without getting bogged down by the things you are “supposed” to do.

The End

Twitter: **@joncalhoun**

Website: **calhoun.io**
(I'll publish the slides here)

Course: **gophercises.com**
(It's FREE!!)

Stickers: **out front by registration?**