Buffalo

Rapid web development in Go

Go Meetup

http:////www.meetup.com/Golang-MTY

Objective

To have a Buffalo application hosted on Heroku

Material

http://github.com/gophers-mty/buffalo-workshop

1. Prework:
Getting started with Go

$GOPATH

The `$GOPATH` is where all Go files must live.

In Go 1.8, the `$GOPATH` defaults to `$HOME/go` if not set explicitly.

```
 $/Users/<username>/go
 $/Users/mayracabrera/go
```

All earlier versions of Go require this environment variable to be set.

Go Workspaces

Under `$GOPATH` there are three folders:

- **bin:** This is where compiled Go programs will be installed
- **pkg:** Compiled package objects live here. You can safely ignored this directory
- **src:** This is where all of your source code for Go projects has to lie

Common Layout

It is common to layout out your Go project in the following directory structure:

```
$GOPATH/src/github.com/<username>/project
```

```
$GOPATH/src/github.com/mayra-cabrera/cats-will-rule-the-
world
```

Exercise:
Create Common Layout

1. Create the three folders inside your `$GOPATH` (`src, bin, & pkg`)

2. Create your username (Github) folder inside the src folder, i.e

```
$GOPATH/src/github.com/mayra-cabrera/
```

Exercise:
Setting up your $PATH

When Go files are installed they are placed into the `$GOPATH/bin` folder. This should be added to your `$PATH` so that these executable files are available to you.

Unix/Mac OS

In your .bash_profile, or equivalent file add (and restart your terminal):

```
export PATH="$GOPATH/bin:$PATH"
```

Exercise: System Check

1. Download the following problem:

[buffalo-workshop/1-prework/system-check.go]()

2. Execute it in your machine:

```
$ go run system-check.go
```

3. If it prints “*Success!*” you’re ready to go!

2. Introduction to Buffalo

Web applications are not simple

- routing
- templating
- database
- assets
- deployment
- testing
- task scripting
- internationalization
- sessions
- cookies
- notifications
- middleware, etc…

Go Standard library?

- ~~routing~~
- ~~templating~~
- ~~database~~
- ~~assets~~
- ~~deployment~~
- testing
- ~~task scripting~~
- ~~internationalization~~
- ~~sessions~~
- cookies
- ~~notifications~~
- ~~middleware, etc~~…

Buffalo to the rescue!

A rapid web development eco-system for Go

Exercise: Installation

Buffalo can be installed with the `go get` command:

```
$ go get -u -v github.com/gobuffalo/buffalo/buffalo
```

Getting around Buffalo

Go to: gobuffalo.io & blog.gobuffalo.io

In your terminal type:

```
$ buffalo --help
```

3. Creating a new application

Few notes before getting started

- You must work within your Go workspace (`$GOPATH`)
- Buffalo assumes you have a database install
- Buffalo won't install Node or NPM for you, but it will install packages (assuming Node/NPM are installed).

Go to your $GOPATH

And type:

```
$ buffalo new hello_world

$ cd hello_world
```

The application

- **actions/app.go:** is where you will configure your application, add routing, middleware, etc
- **database.yml:** Configuration of your database. Supports Postgres, MySQL & Sqlite3
- **model.go:** You will find the connection to your database
- Views inside **templates** folder

Lift the application

Make sure your application works

```
$ buffalo setup
```

Create the database

```
$ buffalo db create
```

Run the application

```
$ buffalo dev
```

Go to localhost:3000 in your browser 👀

Let’s add a route!

On app.go type the following:

```
app.GET("/hello", func(c buffalo.Context) error {
   return c.Render(200, r.String("Hello world!"))
})
```

Let’s display a view!

On app.go type the following:

```
app.GET("/hello-world", func(c buffalo.Context) error {
  return c.Render(200, r.HTML("hello-world.html"))
})
```

Exercises

1. Create a hello_world application

```
$ buffalo new hello_world
```

2. Make sure your applications works

```
$ buffalo setup
```

Exercises

1. Modify the `/hello` handler to change it's greeting based on a query parameter. So it outputs “Hello Mayra” if `/hello?name=Mayra` is requested

2. Modify the `/hello-world` handler to also receive a name parameter.

3. Pass down the parameter on `/hello-world` handler and display it on the view

Reference: **https://gobuffalo.io/docs/routing#parameters**

Solutions

On app.go

```
app.GET("/hello", func(c buffalo.Context) error {
  name := c.Param("name")
  return c.Render(200, r.String("Hello " + name))
})
```

Solutions

On app.go

```
app.GET("/hello-world", func(c buffalo.Context) error {
   name := c.Param("name")
   c.Set("name", name)
   return c.Render(200, r.HTML("hello-world.html"))
})
```

On templates/hello-world.html

```
<div class="content">
 Hello <%= name %>!
</div>
```

4. Working with CRUD’s

CRUD

Create, **R**ead, **U**pdate, **D**elete

Generating Resources

Generate a new application

```
$ buffalo new bloggy
```

Generate a “Post” resource

```
$ buffalo generate resource post title:string
description
```

Run the migrations with

```
$ buffalo db migrate
```

Generating resources

When we ran that command Buffalo generated a lot of files for us:

- A **model** to represent a Post
- **Migrations** for creating the posts table
- Implementations of all the buffalo **resource endpoints** to CRUD a Post model
- **Views** to CRUD a Post model

Exercise

Generate a bloggy application

Generate Post resource with a title & description

Exercise

Generate a User resource:

```
$ buffalo g resource user first_name last_name email
```

Run the migrations

```
$ buffalo db migrate
```

5. Models and Forms

Writing Forms

While forms can be hand coded in Buffalo, it is recommended to use the **github.com/gobuffalo/tags** and its form implementations.

The templating system has built-in helpers to work with this package:

- form - builds a generic form (using Bootstrap)
- form_for - builds a form for a model (using Bootstrap)

Exercise

1. Add a new string field called `category` to `Post.` You can do this with:

```
$ buffalo db g migration add_category_to_post
```

2. Modify Post’s form to include a select with the following options: “Draft” & “Complete”

3. Ensure this field is save on the database

Solutions

1. Migrations:

add_category_to_posts.up.fizz:

```
add_column("posts", "category", "string", {})
```

add_category_to_posts.down.fizz:

```
drop_column("posts", “category")
```

Solutions

2. Modify Post’ files:

templates/posts/_form.html

```
…
<%= f.SelectTag("Category", {options: ["Draft", "Complete"]}) %>
…
<button class="btn btn-success" role="submit">Save</button>
```

Solutions

2. Modify Post' files:

templates/posts/index.html

```
<table class="table table-striped">
  <thead>
  <th>Title</th>
  <th>Category</th>
  <th></th>
  </thead>
  <tbody>
    <%= for (post) in posts { %>
      <tr>
        <td><%= post.Title %></td>
        <td><%= post.Category %></td>
        <td>
          <div class="pull-right">
           …
          </div>
        </td>
      </tr>
    <% } %>
  </tbody>
</table>
```

Solutions

3. Ensure this field is saved on database

models/post.go

```
type Post struct {
    ID          uuid.UUID `json:"id" db:"id"`
    CreatedAt time.Time `json:"created_at" db:"created_at"`
    UpdatedAt time.Time `json:"updated_at" db:"updated_at"`
    Title       string     `json:"title" db:"title"`
    Description string     `json:"description" db:"description"`
    Category    string     `json:"category" db:"category
}
```

Validations

Buffalo includes by default a selection of “common” validators that can easily used:

github.com/markbates/validate/

Validation on the model

Buffalo will attempt to add some default validations based on the types of the model's fields.

```
func (u *User) Validate(tx *pop.Connection) (*validate.Errors, error) {
   return validate.Validate(
      &validators.StringIsPresent{Field: u.FirstName, Name: "FirstName"},
      &validators.StringIsPresent{Field: u.LastName, Name: "LastName"},
      &validators.StringIsPresent{Field: u.Email, Name: "Email"},
   ), nil
}
```

Exercise

1. Add a validation in Post that ensures a post has a description if it’s complete

Solution

1. Migrations:

```
func (p *Post) Validate(tx *pop.Connection) (*validate.Errors, error) {
    rules := validate.Validate(&validators.StringIsPresent{Name: "Name",
Field: p.Title})
    if p.Category == "Complete" {
        rules = validate.Validate(
            &validators.StringIsPresent{Name: "Title", Field: p.Title},
            &validators.StringIsPresent{Name: "Description", Field:
p.Description},
        )
    }
    return rules, nil
}
```

6. Deployment

Setup

1. Head over Heroku and make sure you have installed Heroku command line:

   https://devcenter.heroku.com/articles/heroku-cli#download-and-install

2. Make sure you're login with

```
$ heroku login
```

3. Create an application with

```
$ heroku create
```

Based on: https://blog.gobuffalo.io/deploying-buffalo-to-heroku-with-docker

Setup

4. Buffalo comes with a Dockerfile and a .dockerignore. You can find a personalized Dockerfile for the project right here:

buffalo-workshop/dockerfile/Dockerfile

5. Setting up Heroku

```
$ heroku config:set GO_ENV=production
$ heroku addons:create heroku-postgresql:hobby-dev
```

Based on: https://blog.gobuffalo.io/deploying-buffalo-to-heroku-with-docker

Deployment!

Deploying and running migrations:

```
$ heroku container:login

$ heroku container:push web

$ heroku run bin/app migrate

$ heroku open
```

Thanks!