

Go Meetup: Buffalo

Rapid web development in Go



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

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:

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

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 ignore 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
```

This will make projects available with the go get tool. It will also help readability later.

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...

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

Installation

Buffalo can be installed with the go get command:

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

Getting around Buffalo

```
$ buffalo - -help
```

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

3. Creating a new application

Few notes before getting started

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

Go to your \$GOPATH

```
$ pwd
```

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

```
$ 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

```
$ buffalo setup
```

```
$ buffalo db create
```

```
$ buffalo dev
```

Go to localhost:3000 on your browser 👁👁

Exercise

- Generate a new application (use `--db-type=sqlite3` to use SQLite)
- Run `buffalo setup` to make sure your new application works
- Start the app with `$ buffalo dev`
- Go to <http://localhost:3000>
- Look over the generated files and try to understand how they all fit together

Routing

On app.go:

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

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

Exercise

- Modify the `/hello` handler to change its greeting based on a query parameter. So it outputs “Hello Mayra” if `/hello?name=Mayra` is requested
- Modify the `/hello-world` handler to also receive a name parameter.
- Pass down the parameter on `/hello-world` handler and display it on the view

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">  
    I'm rendering a view!  
</div>
```

```
<%= name %>
```


Working with CRUD's

Generating Resources

- Generate a new application

```
$ buffalo new bloggy
```

- Generate a “Post” resource

```
$ buffalo g resource post title:text
```

- 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 end points to CRUD a Post model
- Views to CRUD a Post model

Exercise

- Generate a User resource:

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

- Run the migrations `$ buffalo db migrate`
- Start the app `$ buffalo dev` and navigate to <http://localhost:3000>
- Play with the forms and pages that were generated
- Run `$ buffalo t routes` to see the routing table

4. Forms and Models

Writing Forms

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

Plush, 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

- Add a new boolean field called `published` to `Post`
- Modify the form to include a checkbox
- Ensure this field is save on the database

My Solution

Validations

The github.com/markbates/validate/validators includes a selection of "common" validators that can easily be used.

Validation on the model

When a new model, or resource, is generated with Buffalo, it will attempt to add some default validations based on the types of the model's fields.

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

Exercise

- Add a validation to `Post` that requires all fields to be present
- Add a validation to `User` to ensure `first_name` and `last_name` are required
- Add a validation to `User` to ensure an `email` is unique

6. Deployment

Setup

Head over Heroku and make sure we have installed heroku command line

Create an application with

```
$ heroku create
```

Build a Dockerfile with a proper .dockerignore

Setting up Heroku

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

Deployment!

Deploying and running migrations

```
$ heroku container:push web  
$ heroku run ./bin/app migrate  
$ heroku open
```


Thanks!

