Getting Started on Heroku with Go

Last updated 15 January 2020

Table of Contents

- Introduction
- Set up
- Prepare the app
- Deploy the app
- View logs
- Define a Procfile
- Scale the app
- Declare app dependencies
- Run the app locally
- Push local changes
- Provision add-ons
- Start a one off dyno
- Define config vars
- Use a database
- Next steps

Introduction

This tutorial will have you deploying a Go app in minutes.

Hang on for a few more minutes to learn how it all works, so you can make the most out of Heroku.

The tutorial assumes that you have:

- a free Heroku account (https://signup.heroku.com/signup/dc).
- Go 1.12+ installed (http://golang.org/doc/install).

Set up

The Heroku CLI requires **Git**, the popular version control system. If you don't already have Git installed, complete the following before proceeding:

- Git installation (https://git-scm.com/book/en/v2/Getting-Started-Installing-Git)
- First-time Git setup (https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup)

In this step you'll install the Heroku Command Line Interface (CLI). You use the CLI to manage and scale your applications, provision add-ons, view your application logs, and run your application locally.

Download and run the installer for your platform:

macOS

Download the installer (https://cli-assets.heroku.com/heroku.pkg)

Also available via Homebrew:

```
$ brew install heroku/brew/heroku
```

Windows

Download the appropriate installer for your Windows installation

64-bit installer (https://cli-assets.heroku.com/heroku-x64

32-bit installer (https://cli-assets.heroku.com/heroku-x86

Ubuntu 16+

Run the following from your terminal:

```
$ sudo snap install heroku --classic
```

Snap is available on other Linux OS's as well (https://snapcraft.io).

Once installed, you can use the `heroku` command from your command shell.

Use the `heroku login` command to log in to the Heroku CLI:

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit
 ›   Warning: If browser does not open, visit
 ›   https://cli-auth.heroku.com/auth/browser/***
heroku: Waiting for login...
Logging in... done
Logged in as me@example.com
```

This command opens your web browser to the Heroku login page. If your browser is already logged in to Heroku, simply click the **`Log in`** button displayed on the page.

This authentication is required for both the `heroku` and `git` commands to work correctly.

If you're behind a firewall that requires use of a proxy to connect with external HTTP/HTTPS services, **you can set the `HTTP_PROXY` or `HTTPS_PROXY` environment variables (https://devcenter.heroku.com/articles/using-the-cli#using-an-http-proxy)** in your local development environment before running the `heroku` command.

Prepare the app

In this step, you will prepare a sample application that's ready to be deployed to Heroku.

If you are new to Heroku, it is recommended that you complete this tutorial using the Heroku-provided sample application.

However, if you have your own existing application that you want to deploy instead, see **this article (https://devcenter.heroku.com/articles/preparing-a-codebase-for-heroku-deployment)** to learn how to prepare it for Heroku deployment.

Clone the sample application so that you have a local version of the code that you can then deploy to Heroku, execute the following commands in your local command shell or terminal:

```
$ git clone https://github.com/heroku/go-getting-started.git
$ cd go-getting-started
```

You now have a functioning git repository that contains a simple application as well as a `go.mod` file, which is used by the Go's module dependency system.

Deploy the app

In this step you will deploy the app to Heroku.

Create an app on Heroku, which prepares Heroku to receive your source code.

```
$ heroku create
Creating polar-inlet-4930... done, stack is heroku-18
https://polar-inlet-4930.herokuapp.com/ | https://git.heroku.com/polar-inlet-4930.git
Git remote heroku added
```

When you create an app, a git remote (called `heroku`) is also created and associated with your local git repository.

Heroku generates a random name (in this case `polar-inlet-4930`) for your app, or you can pass a parameter to specify your own app name.

Now deploy your code:

```
$ git push heroku master
Enumerating objects: 521, done.
Counting objects: 100% (521/521), done.
Delta compression using up to 8 threads
Compressing objects: 100% (309/309), done.
Writing objects: 100% (521/521), 226.26 KiB | 45.25 MiB/s, done.
Total 521 (delta 141), reused 501 (delta 134)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Go app detected
remote: -----> Fetching jq... done
remote:
remote: -----> Detected go modules - go.mod
remote:
remote: -----> Installing go1.12.1
remote: -----> Fetching go1.12.1.linux-amd64.tar.gz... done
remote:  !!    Installing package '.' (default)
remote:  !!
remote:  !!    To install a different package spec add a comment in the following form to your `go.mod` file:
remote:  !!    // +heroku install ./cmd/...
remote:  !!
remote: -----> Running: go install -v -tags heroku -mod=vendor .
remote: gopkg.in/bluesuncorp/validator.v5
remote: github.com/gin-gonic/gin/render
remote: github.com/manucorporat/sse
remote: github.com/mattn/go-colorable
remote: golang.org/x/net/context
remote: github.com/heroku/x/hmetrics
remote: github.com/heroku/x/hmetrics/onload
remote: github.com/gin-gonic/gin/binding
remote: github.com/gin-gonic/gin
remote: github.com/heroku/go-getting-started
remote:
remote: Compiled the following binaries:
remote:        ./bin/go-getting-started
remote:
remote: -----> Discovering process types
remote:        Procfile declares types -> web
remote:
remote: -----> Compressing...
remote:        Done: 5.5M
remote: -----> Launching...
remote:        Released v3
remote:        https://go-on-heroku.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/go-on-heroku.git
 * [new branch]      master -> master
```

The application is now deployed.

Visit the app at the URL generated by its app name.

As a handy shortcut, you can open the website as follows:

```
$ heroku open
```

View logs

Heroku treats logs as streams of time-ordered events aggregated from the output streams of all your app and Heroku components, providing a single channel for all of the events.

View information about your running app using one of the logging commands (https://devcenter.heroku.com/articles/logging), `heroku logs —-tail`:

```
$ heroku logs --tail
2019-03-20T23:50:05.000000+00:00 app[api]: Build started by user edward@heroku.com
2019-03-20T23:50:16.285107+00:00 app[api]: Scaled to web@1:Free by user edward@heroku.com
2019-03-20T23:50:16.269238+00:00 app[api]: Release v3 created by user edward@heroku.com
2019-03-20T23:50:16.269238+00:00 app[api]: Deploy 89f8247e by user edward@heroku.com
2019-03-20T23:50:17.344246+00:00 heroku[web.1]: Starting process with command `go-getting-started`
2019-03-20T23:50:19.124843+00:00 app[web.1]: [GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in production.
2019-03-20T23:50:19.124866+00:00 app[web.1]: - using env:       export GIN_MODE=release
2019-03-20T23:50:19.124868+00:00 app[web.1]: - using code:      gin.SetMode(gin.ReleaseMode)
2019-03-20T23:50:19.124869+00:00 app[web.1]:
2019-03-20T23:50:19.124876+00:00 app[web.1]: [GIN-debug] GET   /static/*filepath         --> github.com/gin-gonic/gin.(*RouterGroup).createStat
2019-03-20T23:50:19.124878+00:00 app[web.1]: [GIN-debug] HEAD  /static/*filepath         --> github.com/gin-gonic/gin.(*RouterGroup).createStat
2019-03-20T23:50:19.124879+00:00 app[web.1]: [GIN-debug] GET   /                         --> main.main.func1 (2 handlers)
2019-03-20T23:50:19.124881+00:00 app[web.1]: [GIN-debug] Listening and serving HTTP on :44768
2019-03-20T23:50:20.124893+00:00 heroku[web.1]: State changed from starting to up
2019-03-20T23:50:34.000000+00:00 app[api]: Build succeeded
2019-03-20T23:53:43.947356+00:00 app[web.1]: [GIN] 2019/03/20 - 23:53:43 | 200 |    1.974798ms | 204.14.239.105 |   GET     /
2019-03-20T23:53:44.159888+00:00 app[web.1]: [GIN] 2019/03/20 - 23:53:44 | 200 |    5.278921ms | 204.14.239.105 |   GET     /static/main.css
2019-03-20T23:53:44.160049+00:00 app[web.1]: [GIN] 2019/03/20 - 23:53:44 | 200 |    4.895772ms | 204.14.239.105 |   GET     /static/lang-logo.p
2019-03-20T23:53:43.948325+00:00 heroku[router]: at=info method=GET path="/" host=floating-caverns-97277.herokuapp.com request_id=d14eb124-9497
2019-03-20T23:53:44.161176+00:00 heroku[router]: at=info method=GET path="/static/lang-logo.png" host=floating-caverns-97277.herokuapp.com requ
2019-03-20T23:53:44.160488+00:00 heroku[router]: at=info method=GET path="/static/main.css" host=floating-caverns-97277.herokuapp.com request_i
2019-03-20T23:53:44.406484+00:00 heroku[router]: at=info method=GET path="/favicon.ico" host=floating-caverns-97277.herokuapp.com request_id=2c
2019-03-20T23:53:44.405506+00:00 app[web.1]: [GIN] 2019/03/20 - 23:53:44 | 404 |       2.419µs | 204.14.239.105 |   GET     /favicon.ico
```

Visit your application in the browser again, and you'll see another log message generated.

Press `Control+C` to stop streaming the logs.

Define a Procfile

Use a Procfile (https://devcenter.heroku.com/articles/procfile), a text file in the root directory of your application, to explicitly declare what command should be executed to start your app.

The `Procfile` in the example app you deployed looks like this:

```
web: bin/go-getting-started
```

This declares a single process type, `web`, and the command needed to run it. The name `web` is important here. It declares that this process type will be attached to the HTTP routing (https://devcenter.heroku.com/articles/http-routing) stack of Heroku, and receive web traffic when deployed. The command used here, `bin/go-getting-started` is the compiled binary of the getting started app. The build process installs compiled binaries into the dyno's `~/bin` directory.

Procfiles can contain additional process types. For example, you might declare one for a background worker process that processes items off of a queue.

Scale the app

Right now, your app is running on a single web dyno (https://devcenter.heroku.com/articles/dynos). Think of a dyno as a lightweight container that runs the command specified in the `Procfile`.

You can check how many dynos are running using the `ps` command:

```
$ heroku ps
=== web (Free): `go-getting-started`
web.1: up 2015/05/12 11:28:21 (~ 4m ago)
```

By default, your app is deployed on a free dyno. Free dynos will sleep after a half hour of inactivity (if they don't receive any traffic). This causes a delay of a few seconds for the first request upon waking. Subsequent requests will perform normally. Free dynos also consume from a monthly, account-level quota of free dyno hours (https://devcenter.heroku.com/articles/free-dyno-hours) - as long as the quota is not exhausted, all free apps can continue to run.

To avoid dyno sleeping, you can upgrade to a hobby or professional dyno type as described in the Dyno Types (https://devcenter.heroku.com/articles/dyno-types) article. For example, if you migrate your app to a professional dyno, you can easily scale it by running a command telling Heroku to execute a specific number of dynos, each running your web process type.

Scaling an application on Heroku is equivalent to changing the number of dynos that are running. Scale the number of web dynos to zero:

```
$ heroku ps:scale web=0
```

Access the app again by hitting refresh on the web tab, or `heroku open` to open it in a web tab. You will get an error message because you no longer have any web dynos available to serve requests.

Scale it up again:

```
$ heroku ps:scale web=1
```

For abuse prevention, scaling a non-free application to more than one dyno requires account verification.

Declare app dependencies

Heroku recognizes an app as being written in Go by the existence of a `go.mod` file in the root directory.

> Heroku also supports **govendor (https://devcenter.heroku.com/articles/go-dependencies-via-govendor)**, **godep (https://devcenter.heroku.com/articles/go-dependencies-via-godep)** & **GB (https://devcenter.heroku.com/articles/go-dependencies-via-gb)**, but this tutorial focuses only on **Go modules (https://github.com/golang/go/wiki/Modules)**.

The demo app you deployed already has a `go.mod` file, and it looks something like this (https://github.com/heroku/go-getting-started/blob/master/go.mod):

```
$ cat go.mod
module github.com/heroku/go-getting-started

go 1.12

require (
        github.com/gin-gonic/gin v0.0.0-20150626140855-4cc2de6207f4
        github.com/heroku/x v0.0.0-20171004170240-705849e307dd
        github.com/manucorporat/sse v0.0.0-20150604091100-c142f0f1baea // indirect
        github.com/mattn/go-colorable v0.0.0-20150625154642-40e4aedc8fab // indirect
        github.com/mattn/go-isatty v0.0.0-20150814002629-7fcbc72f853b // indirect
        github.com/stretchr/testify v1.3.0 // indirect
        golang.org/x/net v0.0.0-20150629084131-d9558e5c97f8 // indirect
        gopkg.in/bluesuncorp/validator.v5 v5.9.1 // indirect
)
```

The `go.mod` file is used by Go tool (https://golang.org/cmd/go/) and specifies both the dependencies that are required to build your application and the build configuration Heroku should use to compile the application. This Go app has a few dependencies, primarily on Gin, a HTTP web framework.

When an app is deployed, Heroku reads this file, installs an appropriate Go version and compiles your code using `go install .`.

Run the app locally

Running apps locally in your own dev environment requires a little more effort. Go is a compiled language and you must first compile the application:

```
$ go build -o bin/go-getting-started -v .
github.com/mattn/go-colorable
gopkg.in/bluesuncorp/validator.v5
golang.org/x/net/context
github.com/heroku/x/hmetrics
github.com/gin-gonic/gin/render
github.com/manucorporat/sse
github.com/heroku/x/hmetrics/onload
github.com/gin-gonic/gin/binding
github.com/gin-gonic/gin
github.com/heroku/go-getting-started
```

Now start your application locally using the `heroku local` command, which was installed as part of the Heroku CLI:

```
$ heroku local web
forego | starting web.1 on port 5000
...
web.1  | [GIN-debug] Listening and serving HTTP on :5000
```

Just like Heroku, `heroku local` examines the `Procfile` to determine what to run.

Open http://localhost:5000 (http://localhost:5000) with your web browser. You should see your app running locally.

To stop the app from running locally, go back to your terminal window and press `Ctrl` + `C` to exit.

Push local changes

In this step you'll learn how to propagate a local change to the application through to Heroku. As an example, you'll modify the application to add an additional dependency and the code to use it.

Dependencies are managed with the Go tool (https://golang.org/cmd/go/#hdr-Add_missing_and_remove_unused_modules) .

Let's modify the app to use the Blackfriday markdown parser. Since this dependency is not already used by your application we need to tell go to fetch a copy of the dependency:

```
$ go get github.com/russross/blackfriday@v2
go: downloading github.com/russross/blackfriday v2.0.0+incompatible
go: extracting github.com/russross/blackfriday v2.0.0+incompatible
go: downloading github.com/shurcooL/sanitized_anchor_name v1.0.0
go: extracting github.com/shurcooL/sanitized_anchor_name v1.0.0
```

This does 3 things:

1. Downloads `v2` of the Blackfriday module and any of it's dependencies to the module cache (https://golang.org/cmd/go/#hdr-GOPATH_and_Modules).
2. Records the Blackfriday dependency, and its dependencies in `go.mod` .
3. Records a cryptographic sum of Blackfriday and it's dependencies in `go.sum`

After that let's introduce a new route, `/mark` , which will show the HTML rendered by the parser. Modify `main.go` so that it uses Blackfriday by adding `"github.com/russross/blackfriday"` to the list of imports, so it looks something like this:

```
import (
    "log"
    "net/http"
    "os"

    "github.com/gin-gonic/gin"
    _ "github.com/heroku/x/hmetrics/onload"
    "github.com/russross/blackfriday"
)
```

Next, modify the `main()` function to introduce a new route that uses blackfriday (see example (https://github.com/heroku/go-getting-started/blob/blackFriday/main.go)). Add the following after the existing `router.GET()` call:

```
router.GET("/mark", func(c *gin.Context) {
  c.String(http.StatusOK, string(blackfriday.Run([]byte("**hi!**"))))
})
```

Finally, let's recompile and start the program locally to manually test our new endpoint:

```
$ go build -o bin/go-getting-started -v .
$ heroku local
```

Visit your application at the new `/mark` route: http://localhost:5000/mark (http://localhost:5000/mark). You should now see the *textual* representation of the HTML generated from the Markdown: `<p><strong>hi!</strong></p>` .

To finish up, let's deploy the local changes to Heroku.

Almost every deploy of a Go application to Heroku follows the same pattern.

First, make sure that any unused modules have been removed from your application:

```
$ go mod tidy
```

Then, make sure that your build is repeatable and resistant to erosion (https://devcenter.heroku.com/articles/erosion-resistance) by vendoring any new dependencies:

```
$ go mod vendor
```

Next, add any modified or new files to the git repository and commit them:

```
$ git add -A .
$ git commit -m "Markdown demo dependency"
[blackFriday fc791f7] Markdown demo dependency
 21 files changed, 5962 insertions(+)
 create mode 100644 vendor/github.com/russross/blackfriday/.gitignore
 create mode 100644 vendor/github.com/russross/blackfriday/.travis.yml
 create mode 100644 vendor/github.com/russross/blackfriday/LICENSE.txt
 create mode 100644 vendor/github.com/russross/blackfriday/README.md
 create mode 100644 vendor/github.com/russross/blackfriday/block.go
 create mode 100644 vendor/github.com/russross/blackfriday/doc.go
 create mode 100644 vendor/github.com/russross/blackfriday/esc.go
 create mode 100644 vendor/github.com/russross/blackfriday/html.go
 create mode 100644 vendor/github.com/russross/blackfriday/inline.go
 create mode 100644 vendor/github.com/russross/blackfriday/markdown.go
 create mode 100644 vendor/github.com/russross/blackfriday/node.go
 create mode 100644 vendor/github.com/russross/blackfriday/smartypants.go
 create mode 100644 vendor/github.com/shurcooL/sanitized_anchor_name/.travis.yml
 create mode 100644 vendor/github.com/shurcooL/sanitized_anchor_name/LICENSE
 create mode 100644 vendor/github.com/shurcooL/sanitized_anchor_name/README.md
 create mode 100644 vendor/github.com/shurcooL/sanitized_anchor_name/go.mod
 create mode 100644 vendor/github.com/shurcooL/sanitized_anchor_name/main.go
```

Deploy just as you did previously:

```
$ git push heroku master
```

And finally, check that your new code is working:

```
$ heroku open mark
```

Provision add-ons

Add-ons are third-party cloud services that provide out-of-the-box additional services for your application, from persistence through logging to monitoring and more.

By default, Heroku stores 1500 lines of logs from your application. However, it makes the full log stream available as a service - and several add-on providers have written logging services that provide things such as log persistence, search, and email and SMS alerts.

In this step you will provision one of these logging add-ons, Papertrail.

Provision the papertrail (https://devcenter.heroku.com/articles/papertrail) logging add-on:

Provisioning this add-on requires **account verification (https://devcenter.heroku.com/articles/account-verification#when-is-verification-required)**.

```
$ heroku addons:create papertrail
Creating giggling-carefully-3978... done
Adding giggling-carefully-3978 to polar-inlet-4930... done
Setting PAPERTRAIL_API_TOKEN and restarting polar-inlet-4930... done, v5
Welcome to Papertrail. Questions and ideas are welcome (support@papertrailapp.com). Happy logging!
```

To help with abuse prevention, provisioning an add-on requires account verification. If your account has not been verified, you will be directed to visit the verification site (https://heroku.com/verify).

The add-on is now deployed and configured for your application. You can list add-ons for your app like this:

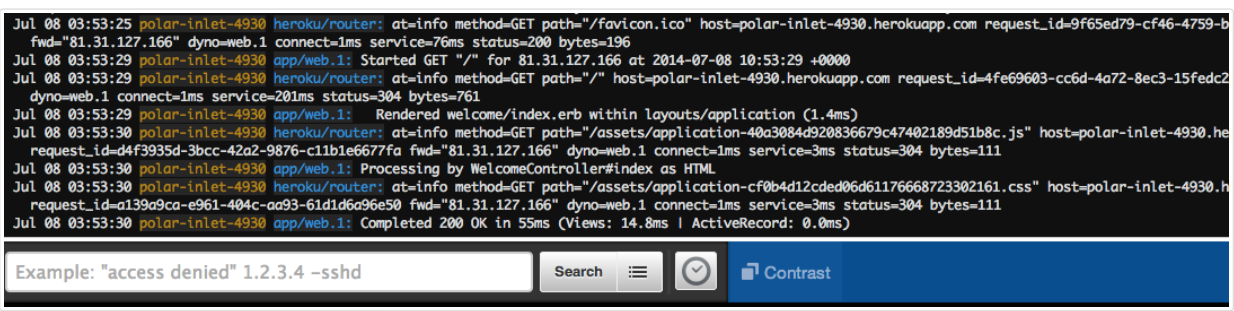
```
$ heroku addons
```

To see this particular add-on in action, visit your application's Heroku URL a few times. Each visit will generate more log messages, which should now get routed to the Papertrail add-on. Visit the Papertrail console to see the log messages:

```
$ heroku addons:open papertrail
```


You may need to wait a few minutes for logs to show up in Papertrail's UI.

Your browser will open up a Papertrail web console, showing the latest log events. The interface lets you search and set up alerts:



Start a one off dyno

You can run a command, typically scripts and applications that are part of your app, in a one-off dyno (https://devcenter.heroku.com/articles/one-off-dynos) using the `heroku run` command. To get a real feel for how dynos work, let's create a one-off dyno that runs the `bash` command, which opens up a shell on that dyno. You can then execute commands there. Each dyno has its own ephemeral filespace, populated with your app and its dependencies - once the command completes (in this case, `bash`), the dyno is removed:

```
$ heroku run bash
Running bash on ⬢ go-getting-started... up, run.9087
~ $ ls
Dockerfile  Makefile  Procfile  README.md  app.json  bin  go.mod  go.sum  heroku.yml  main.go  static  templates  vendor
~ $ exit
exit
```

Don't forget to type `exit` to exit the shell and terminate the dyno.

If you receive an error, `Error connecting to process`, then you may need to configure your firewall (https://devcenter.heroku.com/articles/one-off-dynos#timeout-awaiting-process).

Define config vars

Heroku lets you externalize configuration (https://12factor.net/config), storing data such as encryption keys or external resource addresses in config vars (https://devcenter.heroku.com/articles/config-vars).

At runtime, config vars are exposed as environment variables to the application. Your application is already reading one config var, the `$PORT` config var. `$PORT` is automatically set by Heroku on `web` dynos (https://devcenter.heroku.com/articles/dynos#web-dynos). Let's explore how to use user-set config vars in your Go application.

Modify `main.go` and add a `repeatHandler` function that returns `Hello From Go!` the number of times specified by the value of the `REPEAT` environment variable. Change the file so that it reads like this (https://github.com/heroku/go-getting-started/blob/configVars/main.go):

```
package main

import (
    "bytes"
    "log"
    "net/http"
    "os"
    "strconv"

    "github.com/gin-gonic/gin"
    _ "github.com/heroku/x/hmetrics/onload"
    "github.com/russross/blackfriday"
)

func repeatHandler(r int) gin.HandlerFunc {
    return func(c *gin.Context) {
        var buffer bytes.Buffer
        for i := 0; i < r; i++ {
            buffer.WriteString("Hello from Go!\n")
        }
        c.String(http.StatusOK, buffer.String())
    }
}

func main() {
    port := os.Getenv("PORT")

    if port == "" {
        log.Fatal("$PORT must be set")
    }

    tStr := os.Getenv("REPEAT")
    repeat, err := strconv.Atoi(tStr)
    if err != nil {
        log.Printf("Error converting $REPEAT to an int: %q - Using default\n", err)
        repeat = 5
    }

    router := gin.New()
    router.Use(gin.Logger())
    router.LoadHTMLGlob("templates/*.tmpl.html")
    router.Static("/static", "static")

    router.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.tmpl.html", nil)
    })

    router.GET("/mark", func(c *gin.Context) {
        c.String(http.StatusOK, string(blackfriday.Run([]byte("**hi!**"))))
    })

    router.GET("/repeat", repeatHandler(repeat))

    router.Run(":" + port)
}
```

`heroku local` will automatically set up the environment based on the contents of the `.env` file in your local directory. In the top-level directory of your project there is already a `.env` file that has the following contents:

```
REPEAT=10
```

Recompile the app and run it:

```
$ go build -o bin/go-getting-started -v .
$ heroku local
```

When you access the `/repeat` route on the app at http://localhost:5000/repeat (http://localhost:5000/repeat) you'll see "Hello From Go!" ten times.

To set the config var on Heroku, execute the following:

```
$ heroku config:set REPEAT=10
Setting config vars and restarting polar-inlet-4930... done, v6
REPEAT: 10
```

View the config vars that are set using `heroku config`:

```
$ heroku config
== polar-inlet-4930 Config Vars
PAPERTRAIL_API_TOKEN: erdKhPeeeehIcdfY7ne
REPEAT:               10
```

Deploy the changes to heroku using what you learned in the Push local changes section and try it out by visiting the `/repeat` handler of your application:

```
$ heroku open repeat
```

Use a database

The add-on marketplace (https://elements.heroku.com/addons/categories/data-stores) has a large number of data stores, from Redis and MongoDB providers, to Postgres and MySQL. In this step you will add a free Heroku Postgres Starter Tier dev database to your app.

Add the database:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on ⬢ go-getting-started... free
Created postgresql-curved-22223 as DATABASE_URL
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Use heroku addons:docs heroku-postgresql to view documentation
```

This creates a database and sets the `$DATABASE_URL` environment variable. Listing the config vars for your app will display the value of `$DATABASE_URL`:

```
$ heroku config
=== polar-inlet-4930 Config Vars
DATABASE_URL:               postgres://xx:yyy@host:5432/d8slm9t7b5mjnd
```

Heroku also provides a `pg` command that shows a lot more:

```
$ heroku pg
=== DATABASE_URL
Plan:                  Hobby-dev
Status:                Available
Connections:           0/20
PG Version:            11.2
Created:               2019-03-21 00:59 UTC
Data Size:             7.7 MB
Tables:                0
Rows:                  0/10000 (In compliance)
Fork/Follow:           Unsupported
Rollback:              Unsupported
Continuous Protection: Off
Add-on:                postgresql-vertical-31536
```

This indicates that you have a hobby database (free), running Postgres 11.2 with no tables or data rows.

Let's add a route to the application that will use this database.

Just like the blackfriday module, we need to `go get` Go's postgresql module, `github.com/lib/pq`, before we can use it.

```
$ go get github.com/lib/pq@v1
go: finding github.com/lib/pq v1.0.0
go: downloading github.com/lib/pq v1.0.0
go: extracting github.com/lib/pq v1.0.0
```

Add a `dbFunc` function to the app and register the `/db` route to use it like so (https://github.com/heroku/go-getting-started/blob/libPQ/main.go):

```
package main

import (
    "bytes"
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"

    "github.com/gin-gonic/gin"
    _ "github.com/heroku/x/hmetrics/onload"
    _ "github.com/lib/pq"
    "github.com/russross/blackfriday"
)

func repeatHandler(r int) gin.HandlerFunc {
    return func(c *gin.Context) {
        var buffer bytes.Buffer
        for i := 0; i < r; i++ {
            buffer.WriteString("Hello from Go!\n")
        }
        c.String(http.StatusOK, buffer.String())
    }
}

func dbFunc(db *sql.DB) gin.HandlerFunc {
    return func(c *gin.Context) {
        if _, err := db.Exec("CREATE TABLE IF NOT EXISTS ticks (tick timestamp)"); err != nil {
            c.String(http.StatusInternalServerError,
                fmt.Sprintf("Error creating database table: %q", err))
            return
        }

        if _, err := db.Exec("INSERT INTO ticks VALUES (now())"); err != nil {
            c.String(http.StatusInternalServerError,
                fmt.Sprintf("Error incrementing tick: %q", err))
            return
        }

        rows, err := db.Query("SELECT tick FROM ticks")
        if err != nil {
            c.String(http.StatusInternalServerError,
                fmt.Sprintf("Error reading ticks: %q", err))
            return
        }

        defer rows.Close()
        for rows.Next() {
            var tick time.Time
            if err := rows.Scan(&tick); err != nil {
                c.String(http.StatusInternalServerError,
                    fmt.Sprintf("Error scanning ticks: %q", err))
                return
            }
            c.String(http.StatusOK, fmt.Sprintf("Read from DB: %s\n", tick.String()))
        }
    }
}

func main() {
    port := os.Getenv("PORT")

    if port == "" {
        log.Fatal("$PORT must be set")
    }

    tStr := os.Getenv("REPEAT")
    repeat, err := strconv.Atoi(tStr)
    if err != nil {
        log.Printf("Error converting $REPEAT to an int: %q - Using default\n", err)
        repeat = 5
    }

    db, err := sql.Open("postgres", os.Getenv("DATABASE_URL"))
    if err != nil {
        log.Fatalf("Error opening database: %q", err)
    }

    router := gin.New()
    router.Use(gin.Logger())
    router.LoadHTMLGlob("templates/*.tmpl.html")
    router.Static("/static", "static")

    router.GET("/", func(c *gin.Context) {
        c.HTML(http.StatusOK, "index.tmpl.html", nil)
    })

    router.GET("/mark", func(c *gin.Context) {
        c.String(http.StatusOK, string(blackfriday.Run([]byte("**hi!**"))))
    })

    router.GET("/repeat", repeatHandler(repeat))

    router.GET("/db", dbFunc(db))

    router.Run(":" + port)
}
```

Update your dependencies, commit the new code, and deploy your changes to Heroku:

```
$ go mod tidy
$ go mod vendor
$ git add -A .
$ git commit -m "/db"
$ git push heroku master
$ heroku open db
```

Reload the page a few times and, you will see something like this:

```
Read from DB: 2019-03-21 01:08:21.671103 +0000 +0000
Read from DB: 2019-03-21 01:08:22.988895 +0000 +0000
Read from DB: 2019-03-21 01:08:23.606315 +0000 +0000
Read from DB: 2019-03-21 01:08:24.111442 +0000 +0000
Read from DB: 2019-03-21 01:08:24.287767 +0000 +0000
Read from DB: 2019-03-21 01:08:24.434537 +0000 +0000
Read from DB: 2019-03-21 01:08:24.582809 +0000 +0000
```

If you have Postgres installed locally (https://devcenter.heroku.com/articles/heroku-postgresql#local-setup), you can use the `heroku pg:psql` command to connect to the remote database and see all the rows:

```
$ heroku pg:psql
--> Connecting to postgresql-vertical-31536
psql (11.2)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

floating-caverns-97277::DATABASE=> SELECT * FROM ticks;
            tick
----------------------------
 2019-03-21 01:08:21.671103
 2019-03-21 01:08:22.988895
 2019-03-21 01:08:23.606315
 2019-03-21 01:08:24.111442
 2019-03-21 01:08:24.287767
 2019-03-21 01:08:24.434537
 2019-03-21 01:08:24.582809
...
```

Read more about Heroku PostgreSQL (https://devcenter.heroku.com/articles/heroku-postgresql).

A similar technique can be used to install MongoDB or Redis add-ons (https://elements.heroku.com/addons/categories/data-stores).

Next steps

You now know how to deploy a Go application, change its configuration, view logs, scale, and attach and use add-ons.

Here's some recommended reading:

- Read How Heroku Works (https://devcenter.heroku.com/articles/how-heroku-works) for a technical overview of the concepts you'll encounter while writing, configuring, deploying and running applications.
- Visit the Go category (https://devcenter.heroku.com/categories/go-support) to learn more about developing and deploying Go applications.