Gopherpad Documentation

Release 1.0

Dalton Hubble

Sep 27, 2017

Contents

CHAPTER 1

Introduction

Foreword

In this tutorial series, we'll build a well structured Go web project called Gopherpad together. The project will start small, proceed incrementally, and pause at points to explain the decisions being made instead of prescribing a final project template.

The general aim of the project is to allow users to write some notes and store them in the cloud in some way. Recongnize that this overall mission is underspecified[1], but this will allow us to raise our ambitions as we complete each phase.

Prerequisites

This tutorial series is suitable for developers with basic Go knowledge and some web programming experience. You should have Go installed and have your Go workspace setup correctly. You should probably complete the Go Tour and read about Packages or have equivalent knowledge.

Familiarity with web frameworks is helpful but not required. Comparisons are made to Django or Flask at a few points, which some readers may appreciate.

Design Goals

Small libraries over Frameworks

Using the standard library and small, well-chosen packages instead of a single framework that attempts to do everything keeps our application lean and flexible for customization[2]. The struct components choosen from community

[1] Many real world engineering problems are underspecified.

[2] Go's excellent modularity facilitates composition of small packages and small, simple interfaces. There is no reason to recreate a Django or Rails style monolithic web framework.

packages will implement very general Go interfaces, so they can be swapped easily if you find or build a nicer implementation.

Modular, Re-usable Components

The project will consist of modular packages for large components (e.g. website, analytics, billing) and for re-usable subcomponents (e.g. login, frontend, api)[3]. Larger components can flexibly be built into separate executables or bundled together as needed. Subcomponents can be moved or open sourced as top level packages easily. Best of all, this is accomplished using standard Go packaging, nothing custom needs to be invented.

Platform Independence

The project will be designed to run as a stand-alone Go project and on virtual platforms and infrastructure. Platform-specific resources and services (e.g. database, email) will be accessed through abstract client libraries that can be swapped to maintain portability. Later, this will allow the project to be easily containerized for deployment on different container platforms.

Note: Gopherpad uses the Google Datastore database for scalability, but this currently introduces an undesirable dependency on App Engine. A protobuf Go client library is in the works, but not ready yet. If you wish, you may follow this tutorial and run the project stand-alone until the database is introduced, at which point you can make your own choice.

---

[3] If you're familiar with Django, this is the same concept behind having a project with multiple re-usable apps included in the source or via INSTALLED_APPS.

CHAPTER 2

Minimal Server

Getting Started

Let's start by making a minimal web server in a single executable (main) Go package and running it. We'll use the standard library's net/http package to create a Server and register Handlers for returning simple HTTP responses.

Create a directory `github.com/username/gopherpad` inside your `$GOPATH/src`. This will serve as the Gopherpad project directory and as the Go *main* package implementing the initial website.

The net/http package exposes a Server struct which has fields Addr and Handler (among others) and can start listening for incoming requests over TCP when `server.ListenAndServe()` is invoked. An Addr is a string representing a TCP network address and a Handler is any object that has a `ServeHTTP(ResponseWriter, *Request)` method to implicitly satisfy the Handler interface.

Create an `app.go` file to start a Server with a single Handler.

```
package main

import (
    "fmt"
    "net/http"
)

// main starts serving the web application
func main() {
    srv := &http.Server{
        Addr:    "localhost:8080",
        Handler: helloHandler{},
    }
    srv.ListenAndServe()
}

// helloHandler writes a greeting
type helloHandler struct{}
```

```
func (h helloHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello World")
}
```

Since the source code is in a single file, you can use 'go run' to compile the file to a temporary directory and run it:

```
$ cd gopherpad
$ go run app.go
```

A more typical approach is to install the executable and run it. An executable package builds an executable command with the name of the package directory (i.e. gopherpad).

```
$ cd gopherpad
$ go install .
$ gopherpad      // $GOPATH/bin should be on your PATH
```

Open localhost:8080 to check the 'Hello World' result. Ctrl+C to stop the process.

Notice that with a single Handler, every path (e.g. /foo, /bar) routes to the same Handler. We'll address that limitation with a ServeMux.

URL Routing: ServeMux

The net/http package exports ServeMux, a simple HTTP request multiplexer to support URL routing. ServeMux allows pattern, Handler pairs to be registered. It implements `ServeHTTP(ResponseWriter, *Request)` by matching a request URL against the closest registered pattern string and calling through to the corresponding Handler. This makes a ServeMux a type of Handler our Server can use.

net/http provides a global `DefaultServeMux` instance for convenience. URL patterns and Handlers can be registered on it via `http.Handle(pattern string, handler Handler)` or `http.HandleFunc(pattern string, handler func(ResponseWriter, *Request))`. The later option allows the handler to be a function, which eliminates the need for the empty struct used previously.

Update app.go:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

// init registers patterns and handlers on DefaultServeMux
func init() {
    http.HandleFunc("/", helloHandler)
    http.HandleFunc("/notes", noteHandler)
}

// main starts serving the web application
func main() {
    address := "localhost:8080"
    log.Printf("Starting Server listening on %s\n", address)
    // create and start Server with DefaultServeMux Handler
    err := http.ListenAndServe(address, nil)
    if err != nil {
```

```
        log.Fatal("ListenAndServe error: ", err)
    }
}

// helloHandler writes a greeting
func helloHandler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "Hello World")
}

// noteHandler writes a HTML textarea
func noteHandler(writer http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(writer, "<h1>Gopherpad</h1><textarea></textarea>")
}
```

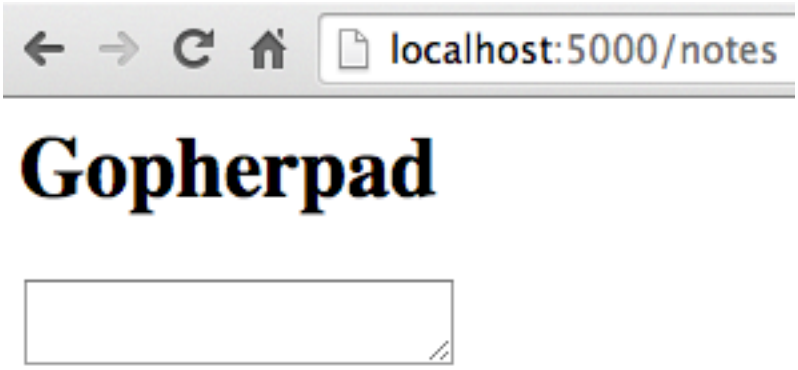
Note: The standard library's ServeMux accepts simple, fixed URL patterns. When variable parts in URL patterns are needed, in *Pattern Routing*, we'll see how to easily swap in a different multiplexing handler for that purpose.

This registers the pattern "/notes" to the `noteHandler` function and the default pattern "/" to the `helloHandler` (which is now a function instead of a struct) onto the global `http.DefaultServeMux`. Each Go source file can have an `init` function to initialize declarations and state as is used here.

The executable's main function has changed to use the net/http library's http.ListenAndServe(addr string, handler Handler) utility function which creates a Server with the given Address and Handler and starts it listening for incoming requests. Passing a nil Handler argument, indicates that the `http.DefaultServeMux` should be used as the Handler.

Re-run the server and localhost:8080/notes will show a simple HTML textarea, while other paths show the default "Hello World" message as before.

```
$ cd gopherpad
$ go install .
$ gopherpad
2014/11/27 16:38:57 Starting Server listening on localhost:8080
```



Each HTML response is hardcoded in a handler, a limitation that will be addressed with *Templates* later.

CHAPTER 3

Pattern Routing

Previously we created an executable package to register two handlers at fixed url patterns and start a simple HTTP server with an Address and Handler.

In this section, we'll start to use a ServeMux that allows variable parts in url patterns and stub out the routes we'd like the application to have.

Choosing a Muxer

Many web apps handle requests to URLs with variable parts such as `/settings/godric` or `/notes/42`. ServeMux handles HTTP requests by multiplexing over handlers registered to fixed patterns only, but remember that ServeMux is just one implementation of the Handler interface. net/http wisely defines the foundational Server and Request, but leaves handling to anything that implements `ServeHTTP(ResponseWriter, *Request)`.

Several HTTP muxers (commonly called 'routers') allow patterns with variable parts (called 'parameters'). For Gopherpad, we seek the following features in a muxer:

- Patterns with parameters and easy retrieval of their values
- HTTP method matching rules
- Route reversal to build redirection URLs from a handler

and we would prefer the muxer provide drop-in compatability with ServeMux.

- bmizerany/pat - minimalistic, does not provide URL reversal
- gorilla/pat - adds URL reversal to bmizerany/pat
- gorilla/mux - feature rich rule matching, more complex
- julienschmidt/httprouter - speedy, has some route limitations

For Gopherpad, we'll use gorilla/pat since it is the simplest library with the desired features.

Note: I'll mention the existence of dghubble/warp (by the author) which offers the features above, a pat-style interface, and (unlike the others) **is** a drop-in compatible fork of http.ServeMux.

Add the `github.com/gorilla/pat` import to `app.go`. Instead of registering pattern, handler pairs on http's DefaultServeMux, register them on a declared pat Router.

```
import (
    ...
    "github.com/gorilla/pat"
)

// ServeMux node
var mux *pat.Router = pat.New()

func init() {
    mux.Get("/notes", noteHandler)
    mux.Get("/", helloHandler)
}

func main() {
    ...
    err := http.ListenAndServe(address, mux)
    ...
}
```

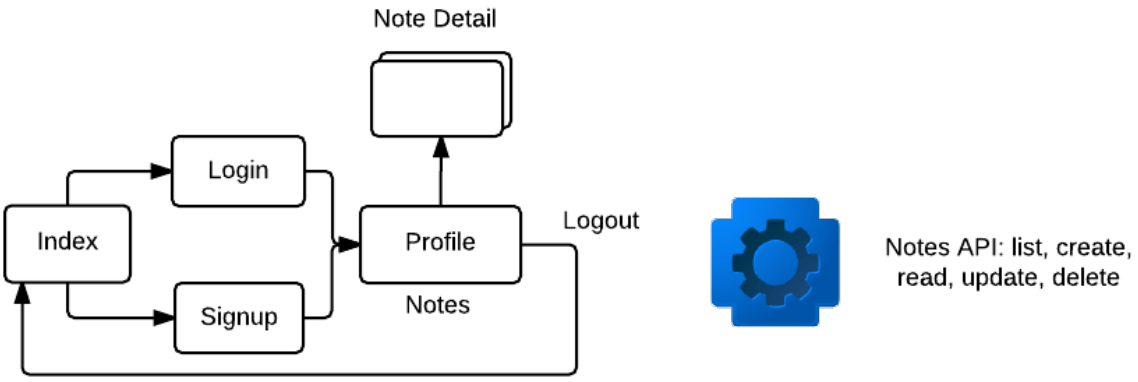
gorilla/pat provides registration methods like `Get(pattern string, h http.HandlerFunc)` (and similar for Post, Put, Delete) instead of ServeMux's `Handle(pattern string, handler Handler)` and `HandleFunc`. Re-run the server and see that it works as it did before, except the handlers will only be called for GET requests.

Notice that we registered "/notes" before the catch all "/". That's because pat checks patterns in the order they are registered, unlike ServeMux. Muxers, even limited to just fixed patterns, can have subtle differences in how URLs match to patterns (typically in regard to priority ordering or slash redirection). There are further quirks in pat, mux, and httprouter due to their different ideas about parameters (e.g. should `/api/v1/woops` match `/{username}`? if so, what is the value of username?). A good approach is to consider ServeMux's behavior as the standard where applicable and check how your muxer behaves.

Now let's replace the toy routes `helloHandler` and `noteHandler` and take advantage of pat's pattern parameters.

Route Design

Gopherpad will have routes for users to signup, login, and logout as well as a profile page for each user showing notes and a note detail page for viewing an individual note.



It will also have a tiny note API under `/api/v1`, supporting the list, create, read, update, and delete actions.

HTTP Verb	Path	Action	Description
GET	/notes	list	list multiple notes
POST	/notes	create	create a note
GET	/notes/{id}	read	get a note
PUT	/notes/{id}	update	update a note
DELETE	/notes/{id}	delete	delete a note

Note: If you're worried (and you should be) about maintaining all these routes in `app.go`, you're absolutely right. What about testable MVC components? Leveraging re-usable components to avoid reimplementing user flow? Splitting the API into a separate executable easily? Refactoring the *Project Structure* is the topic of the next section.

Adding Handlers

Add stub handler functions for the Gopherpad routes for note browsing, login flow, and the API.

```
// ServeMux node
var mux *pat.Router = pat.New()

// init registers patterns and handlers on mux
func init() {
    // api
    mux.Post("/api/v1/notes", noteCreate)
    mux.Get("/api/v1/notes/{id}", noteRead)
    mux.Put("/api/v1/notes/{id}", noteUpdate)
    mux.Delete("/api/v1/notes/{id}", noteDelete)
    // login
    mux.Get("/signup", signupHandler)
    mux.Get("/login", loginHandler)
    mux.Get("/logout", logoutHandler)
    // frontend
    mux.Get("/notes/{id}", noteDetailHandler)
    mux.Get("/profile", profileHandler)
    // default
    mux.Get("/", defaultHandler)
}
```

```
// main starts serving the web application
func main() {
    ...
}

func signupHandler(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "signup")
}

func loginHandler(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "login")
}

func logoutHandler(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "logout")
}

func profileHandler(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "profile")
}

func noteDetailHandler(w http.ResponseWriter, request *http.Request) {
    noteId := request.URL.Query().Get(":id")
    fmt.Fprintf(w, "note %s detail", noteId)
}

func defaultHandler(w http.ResponseWriter, request *http.Request) {
    http.NotFound(w, request)
}

// API

func noteList(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "note list")
}

func noteCreate(w http.ResponseWriter, request *http.Request) {
    fmt.Fprintf(w, "note create")
}

func noteRead(w http.ResponseWriter, request *http.Request) {
    noteId := request.URL.Query().Get(":id")
    fmt.Fprintf(w, "read note %s", noteId)
}

func noteUpdate(w http.ResponseWriter, request *http.Request) {
    noteId := request.URL.Query().Get(":id")
    fmt.Fprintf(w, "update note %s", noteId)
}

func noteDelete(w http.ResponseWriter, request *http.Request) {
    noteId := request.URL.Query().Get(":id")
    fmt.Fprintf(w, "delete note %s", noteId)
}
```

Now for routes with an `{id}` parameter, gorilla/pat will capture the value from the URL and make it available in the URL Values map, accessible through `someURL.Query()`.

Re-run the server and visit the following URLs to check that the GET routes return the expected message:

- http://localhost:8080/signup
- http://localhost:8080/login
- http://localhost:8080/logout
- http://localhost:8080/profile
- http://localhost:8080/notes/3
- http://localhost:8080/api/v1/notes/3

and curl the POST, PUT, and DELETE routes:

```
$ curl -X POST http://localhost:8080/api/v1/notes      # note create
$ curl -X PUT http://localhost:8080/api/v1/notes/3     # update note 3
$ curl -X DELETE http://localhost:8080/api/v1/notes/3  # delete note 3
```

We'll add tests to do these checks automatically in *Routing Tests*, but first let's refactor the *Project Structure* to meet some of our original design goals.

CHAPTER 4

Project Structure

CHAPTER 5

Templates

CHAPTER 6

Routing Tests