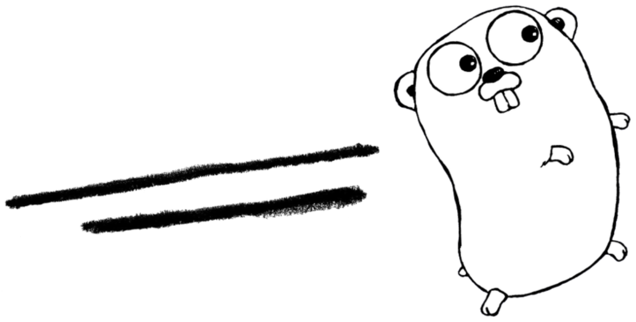
Eamonn McEvoy Follow
Software Engineer in Amsterdam
Feb 6 · 7 min read



Make yourself a Go web server with MongoDb. Go on, Go on, Go on…

I mostly program in c# and node.js, this article is a result of my recent dabbling in go. I set out to create a simple web back end in go to see how it compares with node. I like node because it allows me to create apis extremely quickly, but on the downside most projects depend heavily on external libraries leading to huge code bloat and potentially malicious code.

I have created a small app to demonstrate setting up a rest api, testing, hashing user passwords, and storing data in MongoDb, the full code can be viewed here:

eamonnmcevoy/go_rest_api

go_rest_api - Go web server with mongoDb

github.com

1. Create and retrieve users from mongoDb
2. Salted passwords
3. REST API—creating and retrieving users
4. The cmd folder—link it all up
5. Authentication (sort of)

· · ·

Package layout

In this project I have tried to follow the 'Standard package layout' described by Ben Johnson.

Here's what this looks like:

```
go_rest_api
  /cmd
    /app
      -app.go
      -main.go
  /pkg
    /mongo
    /server
    /mock
    -user.go
```

In the `/cmd` folder we have our main application code, this is simply used to wire up our dependencies and start the server.

The `/pkg` folder is where the bulk of our code lives. At the root level we define our domain types and interfaces, our root package will not depend on any other package.

· · ·

Part 0 — Environment Setup

If you haven't already installed Go, go do that: Install guide

If you haven't already learned the basics of Go, go do that: Go tour

Ensure your GOPATH directory is layed out in this format:

```
GOPATH/
    /bin (to output your built executable files)
    /pkg (to store your projects dependencies)
    /src (your code goes here)
```

We will use a few external dependencies in this project so lets download them all now. Run the following commands in your GOPATH dir.

```
//required
go get gopkg.in/mgo.v2                  //MongoDb driver
go get golang.org/x/crypto/bcrypt       //password hashing


//could live without
go get github.com/gorilla/mux           //http router
go get github.com/gorilla/handlers      //http request
logging
go get github.com/google/uuid           //generate password
salt
```

We could implement this project without Gorilla thanks to Go's fantastic built in http library. However, once your rest api starts to increase in complexity a more fully featured routing toolkit like Gorilla will come in very handy.

Last but not least, install a version of MongoDb and start the server by running `mongod` on the command line.

https://docs.mongodb.com/manual/installation/

. . .

Part 1—Create and retrieve users from Mongo

Lets get started, the first thing we need is a User data type. In the `pkg` folder create the file `user.go` and define the type and service interface.

```go
package root

type User struct {
  Id          string  `json:"id"`
  Username    string  `json:"username"`
  Password    string  `json:"password"`
}

type UserService interface {
```

go_web_server/pkg/user.go

Next we need to implement our `UserService` , since we have decided to use MongoDb as our database lets create a `mongo` package and implement the service in that.

Create a folder `pkg/mongo` , and add two files `user_model.go` , and `user_service.go`

```go
package mongo

import (
  "go_rest_api/pkg"
  "gopkg.in/mgo.v2/bson"
  "gopkg.in/mgo.v2"
)

type userModel struct {
  Id           bson.ObjectId `bson:"_id,omitempty"`
  Username     string
  Password     string
}

func userModelIndex() mgo.Index {
  return mgo.Index{
    Key:        []string{"username"},
    Unique:     true,
    DropDups:   true,
    Background: true,
    Sparse:     true,
  }
}
```

go_web_server/pkg/mongo/user_model.go

In this file we define 3 important things

1. The document structure we will be inserting into Mongo— `userModel` . We are not using `root.Model` here so that we can keep our dependency on `gopkg/mgo.v2/bson` confined to the `mongo` package.
2. The index to be applied to our user collection. This index allows us to quickly query Mongo on the `Username` field and prevents duplicate `Username` 's.
3. `newUserModel` and `toRootUser` functions to convert between `userModel` and `root.User`

You may have noticed the obvious security flaw in this model, we are storing passwords in plain text; Don't worry we will address this with a salted hash in part 2.

```go
package mongo

import (
  "gopkg.in/mgo.v2"
  "gopkg.in/mgo.v2/bson"
  "go_rest_api/pkg"
)

type UserService struct {
 collection *mgo.Collection
}
```

go_web_server/pkg/mongo/user_service.go

This is the basis for our `UserService` implementation. The constructor gets a collection from the `session` parameter and sets up the user index.

```go
func(p *UserService) Create(u *root.User) error {
  user := newUserModel(u)
  return p.collection.Insert(&user)
}

func(p *UserService) GetByUsername(username string) (*root
  model := userModel{}
```

go_web_server/pkg/mongo/user_service.go

Implementing our interface is very simple. The `Create` function is pretty self explanatory, convert `root.User` to `userModel` and insert into our collection. `GetByUsername` performs a Mongo find operation for a single user with a matching `username` .

Lastly, we will create a `session.go` file to manage the connection to Mongo.

```go
package mongo

import (
  "gopkg.in/mgo.v2"
)

type Session struct {
  session *mgo.Session
}

func NewSession(url string) (*Session,error) {
  session, err := mgo.Dial("localhost:27017")
  if err != nil {
    return nil,err
  }
  return &Session{session}, err
}

func(s *Session) Copy() *Session {
  return &Session{s.session.Copy()}
```

go_web_server/pkg/mongo/session.go

This file defines three simple functions to create, access, and close a Mongo session, as well as get a pointer to a collection from that session.

That's it, we have fully implemented our `UserService` as defined in the `root` package, and exported functions for creating and maintaining a Mongo session; time to write some tests and see if it works. Create another new file in the Mongo package `mongo_test.go`

```go
package mongo_test

import (
  "log"
  "testing"
  "go_rest_api/pkg"
  "go_rest_api/pkg/mongo"
)


const (
  mongoUrl = "localhost:27017"
  dbName = "test_db"
  userCollectionName = "user"
)

func Test_UserService(t *testing.T) {
  t.Run("CreateUser", createUser_should_insert_user_into_m
}

func createUser_should_insert_user_into_mongo(t *testing.T
  //Arrange
  session, err := mongo.NewSession(mongoUrl)
  if(err != nil) {
    log.Fatalf("Unable to connect to mongo: %s", err)
  }
  defer session.Close()
  userService := mongo.NewUserService(session.Copy(), dbNa

  testUsername := "integration_test_user"
  testPassword := "integration_test_password"
  user := root.User{
    Username: testUsername,
    Password: testPassword }
```

go_web_server/pkg/mongo/mongo_test.go

In this integration test we insert a single user into our database, verify that 1 user exists in the collection, and that they matches our test data. Head to the command line and run the test

```
➜ go test ./src/go_rest_api/pkg/mongo/
ok      go_rest_api/pkg/mongo    0.145s
```

Success!

…or is it? You'll notice that, if we run the test a second time we get an error.

```
2017/03/14 22:58:04 Unable to create user: E11000 duplicate
key error collection: test_db.user index: username_1 dup
key: { : "integration_test_user" }
```

We need a way to clean up the test database after ourselves, lets extend `session.go` to provide a function to drop a database. Open up `session.go` and add the following function:

```go
func(s *Session) DropDatabase(db string) error {
  if(s.session != nil) {
    return s.session.DB(db).DropDatabase()
  }
  return nil
```

go_web_server/pkg/mongo/session.go

Then modify the defer statement in `mongo_test.go`

```go
defer func() {
    session.DropDatabase(dbName)
    session.Close()
}()
```

go_web_server/pkg/mongo/mongo_test.go

Now we run the tests to our hearts content!

· · ·

Part 2—Salted passwords

In part 1 we implemented a service to store user documents in MongoDb; however we are storing the user password in plain text. Lets introduce a new go package to allow us to generate and compare salted hashes.

Define an interface for our hash functions in the root package:

```go
package root

type Hash interface {
	Generate(s string) (string, error)
	Compare(hash string, s string) error
```

go_web_server/pkg/hash.go

And the implementation, in `/pkg/crypto/hash.go`

```go
package crypto

import (
	"errors"
	"strings"

	"github.com/google/uuid"
	"golang.org/x/crypto/bcrypt"
)

//Hash implements root.Hash
type Hash struct{}

var deliminator = "||"

//Generate a salted hash for the input string
func (c *Hash) Generate(s string) (string, error) {
	salt := uuid.New().String()
	saltedBytes := []byte(s + salt)
	hashedBytes, err := bcrypt.GenerateFromPassword(sal
	if err != nil {
		return "", err
	}

	hash := string(hashedBytes[:])
```

go_web_server/pkg/crypto/hash.go

Our implementation generates a hash with the format `{hash}||{salt}`, this make storing the the hash-salt combination slightly more convenient as we only in a single database field instead of two. We now need to test our implementation to see if it works, we'll test that we can successfully compare a generated hash to an input string, detect when an incorrect attempt is made, and that a generated hash produces a different result each time it is called.

```
package crypto_test

import (
	"go_web_server/pkg/crypto"
	"testing"
)

func Test_Hash(t *testing.T) {
	t.Run("Can hash and compare", should_be_able_to_has
	t.Run("Can detect unequal hashes", should_return_er
	t.Run("Generates a different salt every time", shou
}

func should_be_able_to_hash_and_compare_strings(t *testing
	//Arrange
	c := crypto.Hash{}
	testInput := "testInput"

	//Act
	generatedHash, generateError := c.Generate(testInpu
	compareError := c.Compare(generatedHash, testInput)

	//Assert
	if generateError != nil {
		t.Error("Error generating hash")
	}
	if testInput == generatedHash {
		t.Error("Generated hash is the same as inpu
	}
	if compareError != nil {
		t.Error("Error comparing hash to input")
	}
}

func should_return_error_when_comparing_unequal_hashes(t *
	//Arrange
	c := crypto.Hash{}
	testInput := "testInput"
	testCompare := "testCompare"

	//Act
```

```
        generatedHash, generateError := c.Generate(testInpu
        compareError := c.Compare(generatedHash, testCompar

```

Now that we know our `cryto` package is working lets modify `user_service.go` to store hashed passwords instead of plain text. Extend the `NewUserService` function to accept an instance of `root.Hash`, then use it to generated a hashed password in `Create`.

```
package mongo

import (
        "go_web_server/pkg"

        "gopkg.in/mgo.v2"
        "gopkg.in/mgo.v2/bson"
)

type UserService struct {
        collection *mgo.Collection
        hash       root.Hash
}

func NewUserService(session *Session, dbName string, colle
        collection := session.GetCollection(dbName, collect
        collection.EnsureIndex(userModelIndex())
        return &UserService{collection, hash}
}

func (p *UserService) Create(u *root.User) error {
        user := newUserModel(u)
        hashedPassword, err := p.hash.Generate(user.Passwor
```

So far so good, but the Mongo tests are now broken due to the added parameter. We'll need to create a mock `Hash` implementation in order to maintain the separation between packages. Create a very simple mock in `pkg/mock/hash.go`

```go
package mock

type Hash struct{}

func (h *Hash) Generate(s string) (string, error) {
        return s, nil
}

```

Then, in `mongo_test.go` simply instantiate the `UserService` instance with the mock Hash implementation.

```go
import (
  "go_web_server/pkg/mock"
  ...
)

  ...
  mockHash := mock.Hash{}
```

That's it, in this section we have created a crypto packing to handle password salting, added interaction between packages, and created the first mock implementation

. . .

Part 3 — REST API for creating and retrieving users

The next major component in the system is the http router, we'll be using the Gorrilla mux package to configure our rest endpoints. This package will consist of three files:

```
go_web_server/pkg/server/
      server.go          -- configure router & starts 
listening
      response.go        -- helper functions for sending 
responses
      user_router.go     -- routes for our user service
```

```
package server

import (
	"go_web_server/pkg"
	"log"
	"net/http"
	"os"

	"github.com/gorilla/handlers"
	"github.com/gorilla/mux"
)

type Server struct {
	router *mux.Router
}

func NewServer(u root.UserService) *Server {
	s := Server{router: mux.NewRouter()}
	NewUserRouter(u, s.newSubrouter("/user"))
	return &s
}
```

go_web_server/pkg/server/server.go

The `NewServer` function initialises the server and user subrouter, this allows us to direct all requests beginning with `/user` into the `user_router.go` file. `Start` simply starts our server on port `8080` , we are additionally passing all requests through the gorilla `LoggingHandler` to provide automatic request logging to stdout.

```go
package server

import (
	"encoding/json"
	"errors"
	"go_web_server/pkg"
	"log"
	"net/http"

	"github.com/gorilla/mux"
)

type userRouter struct {
	userService root.UserService
}

func NewUserRouter(u root.UserService, router *mux.Router)
	userRouter := userRouter{u}

	router.HandleFunc("/", userRouter.createUserHandler
	router.HandleFunc("/{username}", userRouter.getUser
	return router
}

func (ur *userRouter) createUserHandler(w http.ResponseWri
	user, err := decodeUser(r)
	if err != nil {
		Error(w, http.StatusBadRequest, "Invalid re
		return
	}

	err = ur.userService.Create(&user)
	if err != nil {
		Error(w, http.StatusInternalServerError, er
		return
	}

	Json(w, http.StatusOK, err)
}

```

go_web_server/pkg/server/user_router.go

The user router exposes two rest endpoints for interacting with the Mongo service.

1. `PUT /users/` for adding new users to the db.
2. `GET /users/{username}` for searching for users with a given username.

You will notice a few unusal function calls to `Json` and `Error`, these are helper functions defined in `response.go`

```
package server

import (
  "net/http"
  "encoding/json"
)

func Error(w http.ResponseWriter, code int, message string
  Json(w, code, map[string]string{"error": message})
}

func Json(w http.ResponseWriter, code int, payload interfa
```

go_web_server/pkg/server/response.go

. . .

Part 4—The `cmd` folder, link it all up

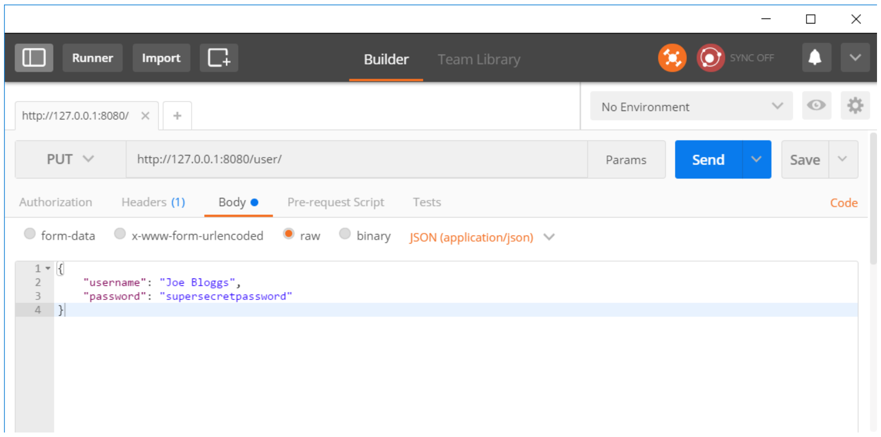
Now that the `server` `mongo` and `crypto` packages are working we can wire them up into a functioning web server. Create the folder `go_web_server/cmd/app` and add `main.go`

```go
package main

import (
	"go_web_server/pkg/crypto"
	"go_web_server/pkg/mongo"
	"go_web_server/pkg/server"
	"log"
)

func main() {
	ms, err := mongo.NewSession("127.0.0.1:27017")
	if err != nil {
		log.Fatalln("unable to connect to mongodb")
	}
	defer ms.Close()
```

Now we can compile and run the server:

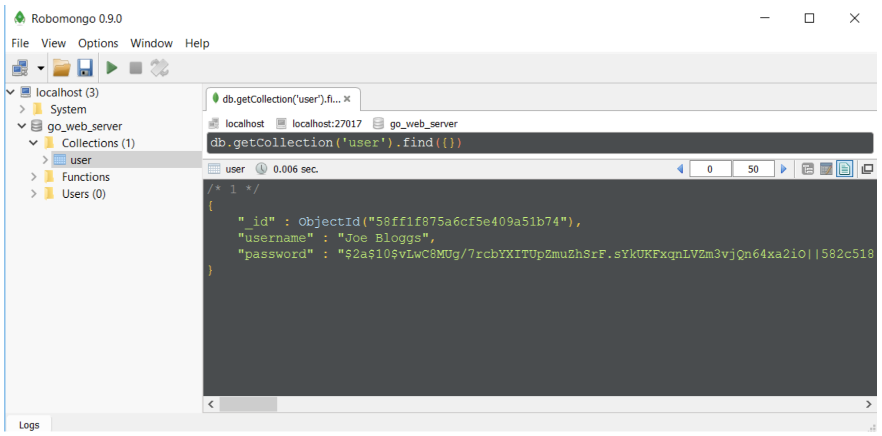
```
➜  go install ./src/go_web_server/cmd/app
➜  ./bin/app.exe
2017/04/25 13:50:35 Listening on port 8080
```

Using postman we can easily test the server endpoints.



PUT http://127.0.0.1:8080/user/

```json
{
    "username": "Joe Bloggs",
    "password": "supersecretpassword"
}
```



```
db.getCollection('user').find({})
```

```
/* 1 */
{
    "_id" : ObjectId("58ff1f875a6cf5e409a51b74"),
    "username" : "Joe Bloggs",
    "password" : "$2a$10$vLwC8MUg/7rcbYXITUpZmuZhSrF.sYkUKFxqnLVZm3vjQn64xa2iO||582c518
}
```

• • •

We now have a working server, at this point we could hook it up to a web front-end or mobile app and develop a user registration form. The next major piece of functionality we need is the ability to authenticate users and protect endpoints. (of course, you shouldn't actually return the users password here).

Part 5—User authentication (sort of)

I won't create a full user authentication system here, just an endpoint to verify user credentials. In my next post I'll show how you can offload this responsibility to the fantastic kong api gateway.

Following the same patterns we used in the previous sections I've added a simple endpoint to accept a username / password combo, and find a user in the database with matching credentials. If a user is found, return them, otherwise return an error.

```go
package root

type Credentials struct {
  Username     string  `json:"username"`
  Password     string  `json:"password"`
}
```

credentials.go hosted with ♡ by GitHub view raw

```go
type UserService interface {
  ...
  Login(c Credentials) (error, User)
}
```

user.go hosted with ♡ by GitHub view raw

```go
func(ur* userRouter) loginHandler(w http.ResponseWriter, r
  log.Println("loginHandler")
  err, credentials := decodeCredentials(r)
  if err != nil {
    Error(w, http.StatusBadRequest, "Invalid request paylo
    return
  }

  var user root.User
  err, user = ur.userService.Login(credentials)
  if err == nil {
    Json(w, http.StatusOK, user)
  } else {
    Error(w, http.StatusInternalServerError, "Incorrect pa
  }
}

...

func decodeCredentials(r *http.Request) (error,root.Creden
  var c root.Credentials
  if r.Body == nil {
    return errors.New("no request body"), c
```

. . .

After all that work we've created a fairly useless api, but its ours and we love it.

I recommend this article if you want to be persuaded to learn go: https://medium.com/@kevalpatel2106/why-should-you-learn-go-f607681fad65

This article to read more about the folder structure I used in this project: https://medium.com/@benbjohnson/standard-package-layout-7cdbc8391fc1