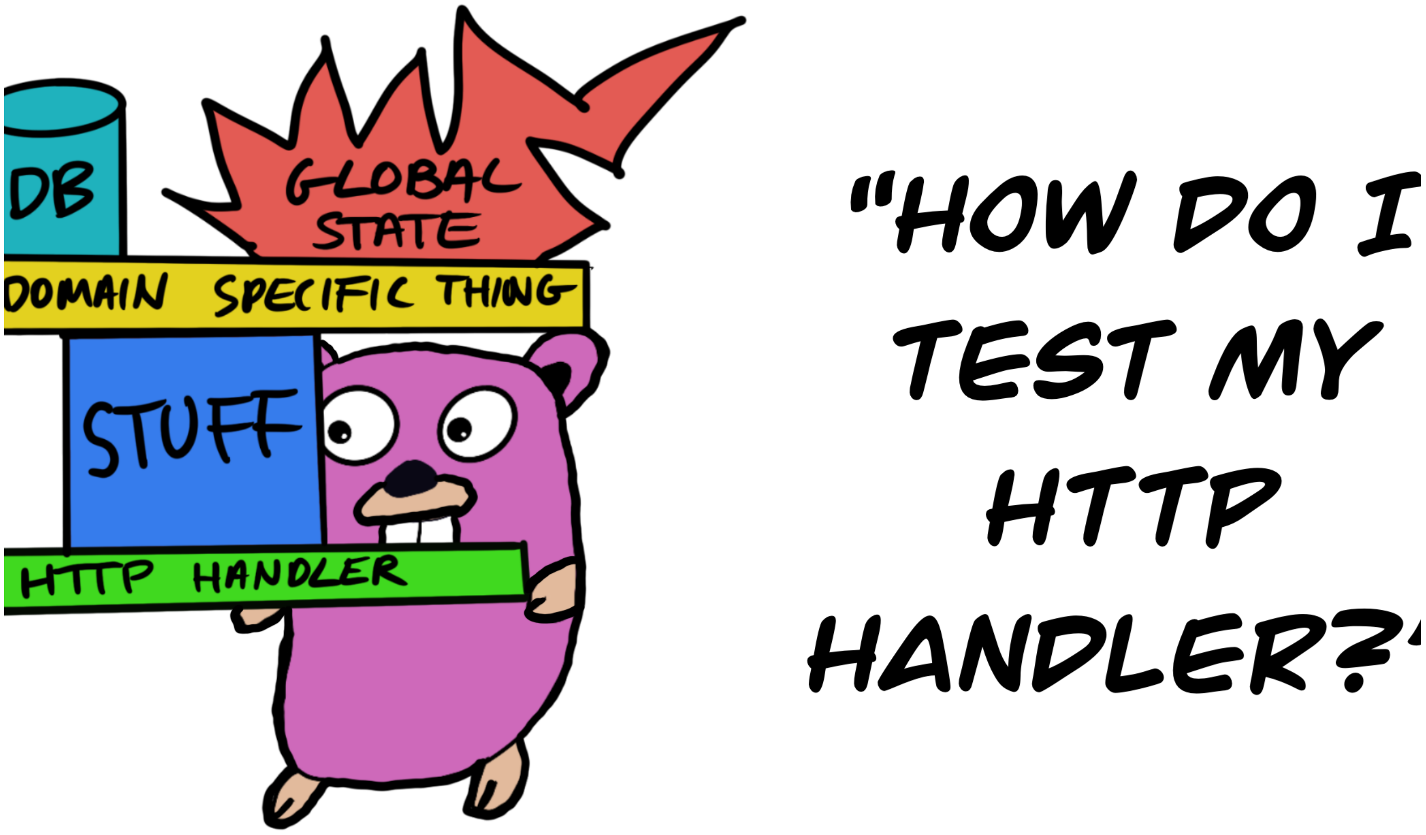
Revisiting HTTP Handlers

You can find all the code here

This book already has a chapter on testing a HTTP handler but this will feature a broader discussion on designing them, so they are simple to test.

We'll take a look at a real example and how we can improve how it's designed by applying principles such as single responsibility principle and separation of concerns. These principles can be realised by using interfaces and dependency injection. By doing this we'll show how testing handlers is actually quite trivial.



Common question in Go community illustrated

Testing HTTP handlers seems to be a recurring question in the Go community, and I think it points to a wider problem of people misunderstanding how to design them.

So often people's difficulties with testing stems from the design of their code rather than the actual writing of tests. As I stress so often in this book:

> If your tests are causing you pain, listen to that signal and think about the design of your code.

An example

Santosh Kumar tweeted me

> How do I test a http handler which has mongodb dependency?

Here is the code

```
func Registration(w http.ResponseWriter, r *http.Request) {
	var res model.ResponseResult
	var user model.User

	w.Header().Set("Content-Type", "application/json")

	jsonDecoder := json.NewDecoder(r.Body)
	jsonDecoder.DisallowUnknownFields()
	defer r.Body.Close()

	// check if there is proper json body or error
	if err := jsonDecoder.Decode(&user); err != nil {
		res.Error = err.Error()
		// return 400 status codes
		w.WriteHeader(http.StatusBadRequest)
		json.NewEncoder(w).Encode(res)
		return
	}

	// Connect to mongodb
	client, _ := mongo.NewClient(options.Client().ApplyURI("mongodb://127.0.0.1:2
	ctx, _ := context.WithTimeout(context.Background(), 10*time.Second)
	err := client.Connect(ctx)
	if err != nil {
		panic(err)
	}
	defer client.Disconnect(ctx)
	// Check if username already exists in users datastore, if so, 400
	// else insert user right away
	collection := client.Database("test").Collection("users")
```

```go
	filter := bson.D{{"username", user.Username}}
	var foundUser model.User

	err = collection.FindOne(context.TODO(), filter).Decode(&foundUser)
	if foundUser.Username == user.Username {
		res.Error = UserExists
		// return 400 status codes
		w.WriteHeader(http.StatusBadRequest)
		json.NewEncoder(w).Encode(res)
		return
	}

	pass, err := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.Defaul
	if err != nil {
		res.Error = err.Error()
		// return 400 status codes
		w.WriteHeader(http.StatusBadRequest)
		json.NewEncoder(w).Encode(res)
		return
	}
	user.Password = string(pass)

	insertResult, err := collection.InsertOne(context.TODO(), user)
	if err != nil {
		res.Error = err.Error()
		// return 400 status codes
		w.WriteHeader(http.StatusBadRequest)
		json.NewEncoder(w).Encode(res)
		return
	}

	// return 200
	w.WriteHeader(http.StatusOK)
	res.Result = fmt.Sprintf("%s: %s", UserCreated, insertResult.InsertedID)
	json.NewEncoder(w).Encode(res)
	return
}
```

Let's just list all the things this one function has to do:

1. Write HTTP responses, send headers, status codes, etc.
2. Decode the request's body into a `User`
3. Connect to a database (and all the details around that)

4. Query the database and applying some business logic depending on the result
5. Generate a password
6. Insert a record

This is too much.

What is a HTTP Handler and what should it do ?

Forgetting specific Go details for a moment, no matter what language I've worked in what has always served me well is thinking about the separation of concerns and the single responsibility principle.

This can be quite tricky to apply depending on the problem you're solving. What exactly *is* a responsibility?

The lines can blur depending on how abstractly you're thinking and sometimes your first guess might not be right.

Thankfully with HTTP handlers I feel like I have a pretty good idea what they should do, no matter what project I've worked on:

1. Accept a HTTP request, parse and validate it.
2. Call some `ServiceThing` to do `ImportantBusinessLogic` with the data I got from step 1.
3. Send an appropriate `HTTP` response depending on what `ServiceThing` returns.

I'm not saying every HTTP handler *ever* should have roughly this shape, but 99 times out of 100 that seems to be the case for me.

When you separate these concerns:

- Testing handlers becomes a breeze and is focused a small number of concerns.
- Importantly testing `ImportantBusinessLogic` no longer has to concern itself with `HTTP`, you can test the business logic cleanly.
- You can use `ImportantBusinessLogic` in other contexts without having to modify it.
- If `ImportantBusinessLogic` changes what it does, so long as the interface remains the same you don't have to change your handlers.

Go's Handlers

`http.HandlerFunc`

> The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers.

`type HandlerFunc func(ResponseWriter, *Request)`

Reader, take a breath and look at the code above. What do you notice?

It is a function that takes some arguments

There's no framework magic, no annotations, no magic beans, nothing.

It's just a function, *and we know how to test functions*.

It fits in nicely with the commentary above:

- It takes a `http.Request` which is just a bundle of data for us to inspect, parse and validate.
- > A `http.ResponseWriter` interface is used by an HTTP handler to construct an HTTP response.

Super basic example test

```go
func Teapot(res http.ResponseWriter, req *http.Request) {
	res.WriteHeader(http.StatusTeapot)
}

func TestTeapotHandler(t *testing.T) {
	req := httptest.NewRequest(http.MethodGet, "/", nil)
	res := httptest.NewRecorder()

	Teapot(res, req)

	if res.Code != http.StatusTeapot {
		t.Errorf("got status %d but wanted %d", res.Code, http.StatusTeapot)
	}
}
```

To test our function, we *call* it.

For our test we pass a `httptest.ResponseRecorder` as our `http.ResponseWriter` argument, and our function will use it to write the `HTTP` response. The recorder will record (or *spy* on) what was sent, and then we can make our assertions.

Calling a `ServiceThing` in our handler

A common complaint about TDD tutorials is that they're always "too simple" and not "real world enough". My answer to that is:

> Wouldn't it be nice if all your code was simple to read and test like the examples you mention?

This is one of the biggest challenges we face but need to keep striving for. It *is possible* (although not necessarily easy) to design code, so it can be simple to read and test if we practice and apply good software engineering principles.

Recapping what the handler from earlier does:

1. Write HTTP responses, send headers, status codes, etc.
2. Decode the request's body into a `User`
3. Connect to a database (and all the details around that)
4. Query the database and applying some business logic depending on the result
5. Generate a password
6. Insert a record

Taking the idea of a more ideal separation of concerns I'd want it to be more like:

1. Decode the request's body into a `User`
2. Call a `UserService.Register(user)` (this is our `ServiceThing`)
3. If there's an error act on it (the example always sends a `400 BadRequest` which I don't think is right), I'll just have a catch-all handler of a `500 Internal Server Error` *for now*. I must stress that returning `500` for all errors makes for a terrible API! Later on we can make the error handling more sophisticated, perhaps with error types.
4. If there's no error, `201 Created` with the ID as the response body (again for terseness/laziness)

For the sake of brevity I won't go over the usual TDD process, check all the other chapters for examples.

New design

```go
type UserService interface {
	Register(user User) (insertedID string, err error)
}

type UserServer struct {
	service UserService
}

func NewUserServer(service UserService) *UserServer {
	return &UserServer{service: service}
}

func (u *UserServer) RegisterUser(w http.ResponseWriter, r *http.Request) {
	defer r.Body.Close()

	// request parsing and validation
	var newUser User
	err := json.NewDecoder(r.Body).Decode(&newUser)

	if err != nil {
		http.Error(w, fmt.Sprintf("could not decode user payload: %v", err),
		return
	}

	// call a service thing to take care of the hard work
	insertedID, err := u.service.Register(newUser)

	// depending on what we get back, respond accordingly
	if err != nil {
		//todo: handle different kinds of errors differently
		http.Error(w, fmt.Sprintf("problem registering new user: %v", err), h
		return
	}

	w.WriteHeader(http.StatusCreated)
	fmt.Fprint(w, insertedID)
}
```

Our `RegisterUser` method matches the shape of `http.HandlerFunc` so we're good to go. We've attached it as a method on a new type `UserServer` which contains a dependency on a `UserService` which is captured as an interface.

Interfaces are a fantastic way to ensure our `HTTP` concerns are decoupled from any specific implementation; we can just call the method on the dependency, and we don't have to care *how* a user gets registered.

If you wish to explore this approach in more detail following TDD read the Dependency Injection chapter and the HTTP Server chapter of the "Build an application" section.

Now that we've decoupled ourselves from any specific implementation detail around registration writing the code for our handler is straightforward and follows the responsibilities described earlier.

The tests!

This simplicity is reflected in our tests.

```go
type MockUserService struct {
	RegisterFunc    func(user User) (string, error)
	UsersRegistered []User
}

func (m *MockUserService) Register(user User) (insertedID string, err error) {
	m.UsersRegistered = append(m.UsersRegistered, user)
	return m.RegisterFunc(user)
}

func TestRegisterUser(t *testing.T) {
	t.Run("can register valid users", func(t *testing.T) {
		user := User{Name: "CJ"}
		expectedInsertedID := "whatever"

		service := &MockUserService{
			RegisterFunc: func(user User) (string, error) {
				return expectedInsertedID, nil
			},
		}
		server := NewUserServer(service)

		req := httptest.NewRequest(http.MethodGet, "/", userToJSON(user))
		res := httptest.NewRecorder()

		server.RegisterUser(res, req)

		assertStatus(t, res.Code, http.StatusCreated)

		if res.Body.String() != expectedInsertedID {
			t.Errorf("expected body of %q but got %q", res.Body.String(),
		}

		if len(service.UsersRegistered) != 1 {
			t.Fatalf("expected 1 user added but got %d", len(service.User
		}

		if !reflect.DeepEqual(service.UsersRegistered[0], user) {
			t.Errorf("the user registered %+v was not what was expected %
		}
	})

	t.Run("returns 400 bad request if body is not valid user JSON", func(t *testi
		server := NewUserServer(nil)

		req := httptest.NewRequest(http.MethodGet, "/", strings.NewReader("tr
		res := httptest.NewRecorder()

		server.RegisterUser(res, req)

		assertStatus(t, res.Code, http.StatusBadRequest)
	})

	t.Run("returns a 500 internal server error if the service fails", func(t *tes
		user := User{Name: "CJ"}

		service := &MockUserService{
			RegisterFunc: func(user User) (string, error) {
				return "", errors.New("couldn't add new user")
			},
		}
		server := NewUserServer(service)

		req := httptest.NewRequest(http.MethodGet, "/", userToJSON(user))
		res := httptest.NewRecorder()

		server.RegisterUser(res, req)

```

```
            assertStatus(t, res.Code, http.StatusInternalServerError)
        })

}
```

Now our handler isn't coupled to a specific implementation of storage it is trivial for us to write a `MockUserService` to help us write simple, fast unit tests to exercise the specific responsibilities it has.

What about the database code? You're cheating!

This is all very deliberate. We don't want HTTP handlers concerned with our business logic, databases, connections, etc.

By doing this we have liberated the handler from messy details, we've *also* made it easier to test our persistence layer and business logic as it is also no longer coupled to irrelevant HTTP details.

All we need to do is now implement our `UserService` using whatever database we want to use

```go
type MongoUserService struct {
}

func NewMongoUserService() *MongoUserService {
        //todo: pass in DB URL as argument to this function
        //todo: connect to db, create a connection pool
        return &MongoUserService{}
}

func (m MongoUserService) Register(user User) (insertedID string, err error) {
        // use m.mongoConnection to perform queries
        panic("implement me")
}
```

We can test this separately and once we're happy in `main` we can snap these two units together for our working application.

```go
func main() {
        mongoService := NewMongoUserService()
        server := NewUserServer(mongoService)
        http.ListenAndServe(":8000", http.HandlerFunc(server.RegisterUser))
}
```

A more robust and extensible design with little effort

These principles not only make our lives easier in the short-term they make the system easier to extend in the future.

It wouldn't be surprising that further iterations of this system we'd want to email the user a confirmation of registration.

With the old design we'd have to change the handler *and* the surrounding tests. This is often how parts of code become unmaintainable, more and more functionality creeps in because it's already *designed* that way; for the "HTTP handler" to handle... everything!

By separating concerns using an interface we don't have to edit the handler *at all* because it's not concerned with the business logic around registration.

Wrapping up

Testing Go's HTTP handlers is not challenging, but designing good software can be!

People make the mistake of thinking HTTP handlers are special and throw out good software engineering practices when writing them which then makes testing them challenging.

Reiterating again; **Go's http handlers are just functions**. If you write them like you would other functions, with clear responsibilities, and a good separation of concerns you will have no trouble testing them, and your codebase will be healthier for it