Dependency Injection

You can find all the code for this chapter here

It is assumed that you have read the structs section before as some understanding of interfaces will be needed for this.

There are *a lot* of misunderstandings around dependency injection around the programming community. Hopefully, this guide will show you how

- You don't need a framework
- It does not overcomplicate your design
- It facilitates testing
- It allows you to write great, general-purpose functions.

We want to write a function that greets someone, just like we did in the hello-world chapter but this time we are going to be testing the *actual printing*.

Just to recap, here is what that function could look like

```
1 func Greet(name string) {
2         fmt.Printf("Hello, %s", name)
3 }
```

But how can we test this? Calling `fmt.Printf` prints to stdout, which is pretty hard for us to capture using the testing framework.

What we need to do is to be able to **inject** (which is just a fancy word for pass in) the dependency of printing.

Our function doesn't need to care _where *or* **how** *the printing happens, so we should accept an* **interface_** rather than a concrete type.**

If we do that, we can then change the implementation to print to something we control so that we can test it. In "real life" you would inject in something that writes to stdout.

If you look at the source code of `fmt.Printf` you can see a way for us to hook in

```go
// It returns the number of bytes written and any write error encountered.
func Printf(format string, a ...interface{}) (n int, err error) {
        return Fprintf(os.Stdout, format, a...)
}
```

Interesting! Under the hood `Printf` just calls `Fprintf` passing in `os.Stdout`.

What exactly *is* an `os.Stdout`? What does `Fprintf` expect to get passed to it for the 1st argument?

```go
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
        p := newPrinter()
        p.doPrintf(format, a)
        n, err = w.Write(p.buf)
        p.free()
        return
}
```

An `io.Writer`

```go
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

From this we can infer that `os.Stdout` implements `io.Writer`; `Printf` passes `os.Stdout` to `Fprintf` which expects an `io.Writer`.

As you write more Go code you will find this interface popping up a lot because it's a great general purpose interface for "put this data somewhere".

So we know under the covers we're ultimately using `Writer` to send our greeting somewhere. Let's use this existing abstraction to make our code testable and more reusable.

Write the test first

```
func TestGreet(t *testing.T) {
        buffer := bytes.Buffer{}
        Greet(&buffer, "Chris")

        got := buffer.String()
        want := "Hello, Chris"

        if got != want {
                t.Errorf("got %q want %q", got, want)
        }
}
```

The `Buffer` type from the `bytes` package implements the `Writer` interface, because it has the method `Write(p []byte) (n int, err error)`.

So we'll use it in our test to send in as our `Writer` and then we can check what was written to it after we invoke `Greet`

Try and run the test

The test will not compile

```
./di_test.go:10:7: too many arguments in call to Greet
    have (*bytes.Buffer, string)
    want (string)
```

Write the minimal amount of code for the test to run and check the failing test output

Listen to the compiler and fix the problem.

```
func Greet(writer *bytes.Buffer, name string) {
```

```
2         fmt.Printf("Hello, %s", name)
3 }
```

`Hello, Chris di_test.go:16: got '' want 'Hello, Chris'`

The test fails. Notice that the name is getting printed out, but it's going to stdout.

Write enough code to make it pass

Use the writer to send the greeting to the buffer in our test. Remember `fmt.Fprintf` is like `fmt.Printf` but instead takes a `Writer` to send the string to, whereas `fmt.Printf` defaults to stdout.

```
1 func Greet(writer *bytes.Buffer, name string) {
2         fmt.Fprintf(writer, "Hello, %s", name)
3 }
```

The test now passes.

Refactor

Earlier the compiler told us to pass in a pointer to a `bytes.Buffer`. This is technically correct but not very useful.

To demonstrate this, try wiring up the `Greet` function into a Go application where we want it to print to stdout.

```
1 func main() {
2         Greet(os.Stdout, "Elodie")
3 }
```

`./di.go:14:7: cannot use os.Stdout (type *os.File) as type *bytes.Buffer in argument to Greet`

As discussed earlier `fmt.Fprintf` allows you to pass in an `io.Writer` which we know both `os.Stdout` and `bytes.Buffer` implement.

If we change our code to use the more general purpose interface we can now use it in both tests and in our application.

```
package main

import (
        "fmt"
        "io"
        "os"
)

func Greet(writer io.Writer, name string) {
        fmt.Fprintf(writer, "Hello, %s", name)
}

func main() {
        Greet(os.Stdout, "Elodie")
}
```

More on io.Writer

What other places can we write data to using `io.Writer`? Just how general purpose is our `Greet` function?

The Internet

Run the following

```
package main

import (
        "fmt"
        "io"
        "log"
        "net/http"
```

```
)

func Greet(writer io.Writer, name string) {
        fmt.Fprintf(writer, "Hello, %s", name)
}

func MyGreeterHandler(w http.ResponseWriter, r *http.Request) {
        Greet(w, "world")
}

func main() {
        log.Fatal(http.ListenAndServe(":5000", http.HandlerFunc(MyGreeterHandler)))
}
```

Run the program and go to http://localhost:5000. You'll see your greeting function being used.

HTTP servers will be covered in a later chapter so don't worry too much about the details.

When you write an HTTP handler, you are given an `http.ResponseWriter` and the `http.Request` that was used to make the request. When you implement your server you *write* your response using the writer.

You can probably guess that `http.ResponseWriter` also implements `io.Writer` so this is why we could re-use our `Greet` function inside our handler.

Wrapping up

Our first round of code was not easy to test because it wrote data to somewhere we couldn't control.

Motivated by our tests we refactored the code so we could control *where* the data was written by **injecting a dependency** which allowed us to:

- **Test our code** If you can't test a function *easily*, it's usually because of dependencies hard-wired into a function *or* global state. If you have a global database connection pool for instance that is used by some kind of service layer, it is likely going to be difficult to test and they will be slow to run. DI will motivate you to inject in a database dependency (via an interface) which you can then mock out with something you can control in your tests.
- **Separate our concerns**, decoupling *where the data goes* from *how to generate it*. If you ever feel

like a method/function has too many responsibilities (generating data *and* writing to a db? handling HTTP requests *and* doing domain level logic?) DI is probably going to be the tool you need.

- **Allow our code to be re-used in different contexts** The first "new" context our code can be used in is inside tests. But further on if someone wants to try something new with your function they can inject their own dependencies.

What about mocking? I hear you need that for DI and also it's evil

Mocking will be covered in detail later (and it's not evil). You use mocking to replace real things you inject with a pretend version that you can control and inspect in your tests. In our case though, the standard library had something ready for us to use.

The Go standard library is really good, take time to study it

By having some familiarity with the `io.Writer` interface we are able to use `bytes.Buffer` in our test as our `Writer` and then we can use other `Writer`s from the standard library to use our function in a command line app or in web server.

The more familiar you are with the standard library the more you'll see these general purpose interfaces which you can then re-use in your own code to make your software reusable in a number of contexts.

This example is heavily influenced by a chapter in The Go Programming language, so if you enjoyed this, go buy it!