Reading files

- **You can find all the code for this chapter here**
- Here is a video of me working through the problem and taking questions from the Twitch stream

In this chapter we're going to learn how to read some files, get some data out of them, and do something useful.

Pretend you're working with your friend to create some blog software. The idea is an author will write their posts in markdown, with some metadata at the top of the file. On startup, the web server will read a folder to create some `Post`s, and then a separate `NewHandler` function will use those `Post`s as a datasource for the blog's webserver.

We've been asked to create the package that converts a given folder of blog post files into a collection of `Post`s.

Example data

hello world.md

```
Title: Hello, TDD world!
Description: First post on our wonderful blog
Tags: tdd, go
---
Hello world!

The body of posts starts after the `---`
```

Expected data

```
type Post struct {
	Title, Description, Body string
	Tags                     []string
}
```

Iterative, test-driven development

We'll take an iterative approach where we're always taking simple, safe steps toward our goal.

This requires us to break up our work, but we should be careful not to fall into the trap of taking a "bottom up" approach.

We should not trust our over-active imaginations when we start work. We could be tempted into making some kind of abstraction that is only validated once we stick everything together, such as some kind of `BlogPostFileParser`.

This is *not* iterative and is missing out on the tight feedback loops that TDD is supposed to bring us.

Kent Beck says:

> Optimism is an occupational hazard of programming. Feedback is the treatment.

Instead, our approach should strive to be as close to delivering *real* consumer value as quickly as possible (often called a "happy path"). Once we have delivered a small amount of consumer value end-to-end, further iteration of the rest of the requirements is usually straightforward.

Thinking about the kind of test we want to see

Let's remind ourselves of our mindset and goals when starting:

- **Write the test we want to see**. Think about how we'd like to use the code we're going to write from a consumer's point of view.
- Focus on *what* and *why*, but don't get distracted by *how*.

Our package needs to offer a function that can be pointed at a folder, and return us some posts.

```go
var posts []blogposts.Post
posts = blogposts.NewPostsFromFS("some-folder")
```

To write a test around this, we'd need some kind of test folder with some example posts in it. *There's nothing terribly wrong with this*, but you are making some trade-offs:

- for each test you may need to create new files to test a particular behaviour
- some behaviour will be challenging to test, such as failing to load files
- the tests will run a little slower because they will need to access the file system

We're also unnecessarily coupling ourselves to a specific implementation of the file system.

File system abstractions introduced in Go 1.16

Go 1.16 introduced an abstraction for file systems; the io/fs package.

> Package fs defines basic interfaces to a file system. A file system can be provided by the host operating system but also by other packages.

This lets us loosen our coupling to a specific file system, which will then let us inject different implementations according to our needs.

> On the producer side of the interface, the new embed.FS type implements fs.FS, as does zip.Reader. The new os.DirFS function provides an implementation of fs.FS backed by a tree of operating system files.

If we use this interface, users of our package have a number of options baked-in to the standard library to use. Learning to leverage interfaces defined in Go's standard library (e.g. `io.fs`, `io.Reader`, `io.Writer`), is vital to writing loosely coupled packages. These packages can then be re-used in contexts different to those you imagined, with minimal fuss from your consumers.

In our case, maybe our consumer wants the posts to be embedded into the Go binary rather than files in a "real" filesystem? Either way, *our code doesn't need to care*.

For our tests, the package testing/fstest offers us an implementation of io/FS to use, similar to the tools we're familiar with in net/http/httptest.

Given this information, the following feels like a better approach,

```go
var posts blogposts.Post
posts = blogposts.NewPostsFromFS(someFS)
```

Write the test first

We should keep scope as small and useful as possible. If we prove that we can read all the files in a directory, that will be a good start. This will give us confidence in the software we're writing. We can check that the count of `[]Post` returned is the same as the number of files in our fake file system.

Create a new project to work through this chapter.

- `mkdir blogposts`
- `cd blogposts`
- `go mod init github.com/{your-name}/blogposts`
- `touch blogposts_test.go`

```go
package blogposts_test

import (
	"testing"
	"testing/fstest"
)

func TestNewBlogPosts(t *testing.T) {
	fs := fstest.MapFS{
		"hello world.md":  {Data: []byte("hi")},
		"hello-world2.md": {Data: []byte("hola")},
	}

	posts := blogposts.NewPostsFromFS(fs)

	if len(posts) != len(fs) {
		t.Errorf("got %d posts, wanted %d posts", len(posts), len(fs))
	}
}
```

Notice that the package of our test is `blogposts_test`. Remember, when TDD is practiced well we take a *consumer-driven* approach: we don't want to test internal details because *consumers* don't care about them. By appending `_test` to our intended package name, we only access exported members from our package - just like a real user of our package.

We've imported `testing/fstest` which gives us access to the `fstest.MapFS` type. Our fake file system will pass `fstest.MapFS` to our package.

> A MapFS is a simple in-memory file system for use in tests, represented as a map from path names (arguments to Open) to information about the files or directories they represent.

This feels simpler than maintaining a folder of test files, and it will execute quicker.

Finally, we codified the usage of our API from a consumer's point of view, then checked if it creates the correct number of posts.

Try to run the test

```
./blogpost_test.go:15:12: undefined: blogposts
```

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

The package doesn't exist. Create a new file `blogposts.go` and put `package blogposts` inside it. You'll need to then import that package into your tests. For me, the imports now look like:

```go
import (
	blogposts "github.com/quii/learn-go-with-tests/reading-files"
	"testing"
	"testing/fstest"
)
```

Now the tests won't compile because our new package does not have a `NewPostsFromFS` function, that returns some kind of collection.

```
./blogpost_test.go:16:12: undefined: blogposts.NewPostsFromFS
```

This forces us to make the skeleton of our function to make the test run. Remember not to overthink the code at this point; we're only trying to get a running test, and to make sure it fails as we'd expect. If we skip this step we may skip over assumptions and, write a test which is not useful.

```go
package blogposts

import "testing/fstest"

type Post struct {
}

func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
	return nil
}
```

The test should now correctly fail

```
=== RUN   TestNewBlogPosts
    blogposts_test.go:48: got 0 posts, wanted 2 posts
```

Write enough code to make it pass

We *could* "slime" this to make it pass:

```go
func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
	return []Post{{}, {}}
}
```

But, as Denise Yu wrote:

> Sliming is useful for giving a “skeleton” to your object. Designing an interface and executing logic are two concerns, and sliming tests strategically lets you focus on one at a time.

We already have our structure. So, what do we do instead?

As we've cut scope, all we need to do is read the directory and create a post for each file we encounter. We don't have to worry about opening files and parsing them just yet.

```go
func NewPostsFromFS(fileSystem fstest.MapFS) []Post {
	dir, _ := fs.ReadDir(fileSystem, ".")
	var posts []Post
	for range dir {
		posts = append(posts, Post{})
	}
	return posts
}
```

`fs.ReadDir` reads a directory inside a given `fs.FS` returning `[]DirEntry`.

Already our idealised view of the world has been foiled because errors can happen, but remember now our focus is *making the test pass*, not changing design, so we'll ignore the error for now.

The rest of the code is straightforward: iterate over the entries, create a `Post` for each one and, return the slice.

Refactor

Even though our tests are passing, we can't use our new package outside of this context, because it is coupled to a concrete implementation `fstest.MapFS`. But, it doesn't have to be. Change the argument to our `NewPostsFromFS` function to accept the interface from the standard library.

```go
func NewPostsFromFS(fileSystem fs.FS) []Post {
	dir, _ := fs.ReadDir(fileSystem, ".")
	var posts []Post
	for range dir {
		posts = append(posts, Post{})
	}
	return posts
}
```

Re-run the tests: everything should be working.

Error handling

We parked error handling earlier when we focused on making the happy-path work. Before continuing to iterate on the functionality, we should acknowledge that errors can happen when working with files. Beyond reading the directory, we can run into problems when we open individual files. Let's change our API (via our tests first, naturally) so that it can return an `error`.

```go
func TestNewBlogPosts(t *testing.T) {
	fs := fstest.MapFS{
		"hello world.md":  {Data: []byte("hi")},
		"hello-world2.md": {Data: []byte("hola")},
	}

	posts, err := blogposts.NewPostsFromFS(fs)

	if err != nil {
		t.Fatal(err)
	}

	if len(posts) != len(fs) {
		t.Errorf("got %d posts, wanted %d posts", len(posts), len(fs))
	}
}
```

Run the test: it should complain about the wrong number of return values. Fixing the code is straightforward.

```go
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
	dir, err := fs.ReadDir(fileSystem, ".")
	if err != nil {
		return nil, err
	}
	var posts []Post
	for range dir {
		posts = append(posts, Post{})
	}
	return posts, nil
}
```

This will make the test pass. The TDD practitioner in you might be annoyed we didn't see a failing test before writing the code to propagate the error from `fs.ReadDir`. To do this "properly", we'd need a new test where we inject a failing `fs.FS` test-double to make `fs.ReadDir` return an `error`.

```go
type StubFailingFS struct {
}

func (s StubFailingFS) Open(name string) (fs.File, error) {
	return nil, errors.New("oh no, i always fail")
}
```

```go
// later
_, err := blogposts.NewPostsFromFS(StubFailingFS{})
```

This should give you confidence in our approach. The interface we're using has one method, which makes creating test-doubles to test different scenarios trivial.

In some cases, testing error handling is the pragmatic thing to do but, in our case, we're not doing anything *interesting* with the error, we're just propagating it, so it's not worth the hassle of writing a new test.

Logically, our next iterations will be around expanding our `Post` type so that it has some useful data.

Write the test first

We'll start with the first line in the proposed blog post schema, the title field.

We need to change the contents of the test files so they match what was specified, and then we can make an assertion that it is parsed correctly.

```go
func TestNewBlogPosts(t *testing.T) {
	fs := fstest.MapFS{
		"hello world.md":  {Data: []byte("Title: Post 1")},
		"hello-world2.md": {Data: []byte("Title: Post 2")},
	}
```

```go
        // rest of test code cut for brevity

        got := posts[0]
        want := blogposts.Post{Title: "Post 1"}

        if !reflect.DeepEqual(got, want) {
                t.Errorf("got %+v, want %+v", got, want)
        }
}
```

Try to run the test

```
./blogpost_test.go:58:26: unknown field 'Title' in struct literal of type blogposts.P
```

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

Add the new field to our `Post` type so that the test will run

```go
type Post struct {
        Title string
}
```

Re-run the test, and you should get a clear, failing test

```
=== RUN   TestNewBlogPosts
=== RUN   TestNewBlogPosts/parses_the_post
    blogpost_test.go:61: got {Title:}, want {Title:Post 1}
```

Write enough code to make it pass

We'll need to open each file and then extract the title

```go
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
	dir, err := fs.ReadDir(fileSystem, ".")
	if err != nil {
		return nil, err
	}
	var posts []Post
	for _, f := range dir {
		post, err := getPost(fileSystem, f)
		if err != nil {
			return nil, err //todo: needs clarification, should we totall
		}
		posts = append(posts, post)
	}
	return posts, nil
}

func getPost(fileSystem fs.FS, f fs.DirEntry) (Post, error) {
	postFile, err := fileSystem.Open(f.Name())
	if err != nil {
		return Post{}, err
	}
	defer postFile.Close()

	postData, err := io.ReadAll(postFile)
	if err != nil {
		return Post{}, err
	}

	post := Post{Title: string(postData)[7:]}
	return post, nil
}
```

Remember our focus at this point is not to write elegant code, it's just to get to a point where we have working software.

Even though this feels like a small increment forward it still required us to write a fair amount of code and make some assumptions in respect to error handling. This would be a point where you should talk to your colleagues and decide the best approach.

The iterative approach has given us fast feedback that our understanding of the requirements is incomplete.

`fs.FS` gives us a way of opening a file within it by name with its `Open` method. From there we read the data from the file and, for now, we do not need any sophisticated parsing, just cutting out the `Title: ` text by slicing the string.

Refactor

Separating the 'opening file code' from the 'parsing file contents code' will make the code simpler to understand and work with.

```go
func getPost(fileSystem fs.FS, f fs.DirEntry) (Post, error) {
        postFile, err := fileSystem.Open(f.Name())
        if err != nil {
                return Post{}, err
        }
        defer postFile.Close()
        return newPost(postFile)
}

func newPost(postFile fs.File) (Post, error) {
        postData, err := io.ReadAll(postFile)
        if err != nil {
                return Post{}, err
        }

        post := Post{Title: string(postData)[7:]}
        return post, nil
}
```

When you refactor out new functions or methods, take care and think about the arguments. You're designing here, and are free to think deeply about what is appropriate because you have passing tests. Think about coupling and cohesion. In this case you should ask yourself:

> Does `newPost` have to be coupled to an `fs.File` ? Do we use all the methods and data from this type? What do we *really* need?

In our case we only use it as an argument to `io.ReadAll` which needs an `io.Reader`. So we should loosen the coupling in our function and ask for an `io.Reader`.

```go
func newPost(postFile io.Reader) (Post, error) {
	postData, err := io.ReadAll(postFile)
	if err != nil {
		return Post{}, err
	}

	post := Post{Title: string(postData)[7:]}
	return post, nil
}
```

You can make a similar argument for our `getPost` function, which takes an `fs.DirEntry` argument but simply calls `Name()` to get the file name. We don't need all that; let's decouple from that type and pass the file name through as a string. Here's the fully refactored code:

```go
func NewPostsFromFS(fileSystem fs.FS) ([]Post, error) {
	dir, err := fs.ReadDir(fileSystem, ".")
	if err != nil {
		return nil, err
	}
	var posts []Post
	for _, f := range dir {
		post, err := getPost(fileSystem, f.Name())
		if err != nil {
			return nil, err //todo: needs clarification, should we totall
		}
		posts = append(posts, post)
	}
	return posts, nil
}

func getPost(fileSystem fs.FS, fileName string) (Post, error) {
	postFile, err := fileSystem.Open(fileName)
	if err != nil {
		return Post{}, err
	}
	defer postFile.Close()
	return newPost(postFile)
}
```

```go
func newPost(postFile io.Reader) (Post, error) {
	postData, err := io.ReadAll(postFile)
	if err != nil {
		return Post{}, err
	}

	post := Post{Title: string(postData)[7:]}
	return post, nil
}
```

From now on, most of our efforts can be neatly contained within `newPost`. The concerns of opening and iterating over files are done, and now we can focus on extracting the data for our `Post` type. Whilst not technically necessary, files are a nice way to logically group related things together, so I moved the `Post` type and `newPost` into a new `post.go` file.

Test helper

We should take care of our tests too. We're going to be making assertions on `Posts` a lot, so we should write some code to help with that

```go
func assertPost(t *testing.T, got blogposts.Post, want blogposts.Post) {
	t.Helper()
	if !reflect.DeepEqual(got, want) {
		t.Errorf("got %+v, want %+v", got, want)
	}
}
```

```go
assertPost(t, posts[0], blogposts.Post{Title: "Post 1"})
```

Write the test first

Let's extend our test further to extract the next line from the file, the description. Up until making it pass should now feel comfortable and familiar.

```
func TestNewBlogPosts(t *testing.T) {
        const (
                firstBody = `Title: Post 1
Description: Description 1`
                secondBody = `Title: Post 2
Description: Description 2`
        )

        fs := fstest.MapFS{
                "hello world.md":  {Data: []byte(firstBody)},
                "hello-world2.md": {Data: []byte(secondBody)},
        }

        // rest of test code cut for brevity
        assertPost(t, posts[0], blogposts.Post{
                Title:       "Post 1",
                Description: "Description 1",
        })

}
```

Try to run the test

```
./blogpost_test.go:47:58: unknown field 'Description' in struct literal of type blogp
```

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

Add the new field to `Post`.

```
type Post struct {
        Title       string
        Description string
}
```

The tests should now compile, and fail.

```
=== RUN   TestNewBlogPosts
    blogpost_test.go:47: got {Title:Post 1
        Description: Description 1 Description:}, want {Title:Post 1 Description:Desc
```

Write enough code to make it pass

The standard library has a handy library for helping you scan through data, line by line; `bufio.Scanner`

> Scanner provides a convenient interface for reading data such as a file of newline-delimited lines of text.

```
func newPost(postFile io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postFile)

        scanner.Scan()
        titleLine := scanner.Text()

        scanner.Scan()
        descriptionLine := scanner.Text()

        return Post{Title: titleLine[7:], Description: descriptionLine[13:]}, nil
}
```

Handily, it also takes an `io.Reader` to read through (thank you again, loose-coupling), we don't need to change our function arguments.

Call `Scan` to read a line, and then extract the data using `Text`.

This function could never return an `error`. It would be tempting at this point to remove it from the return type, but we know we'll have to handle invalid file structures later so, we may as well leave it.

Refactor

We have repetition around scanning a line and then reading the text. We know we're going to do this operation at least one more time, it's a simple refactor to DRY up so let's start with that.

```go
func newPost(postFile io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postFile)

        readLine := func() string {
                scanner.Scan()
                return scanner.Text()
        }

        title := readLine()[7:]
        description := readLine()[13:]

        return Post{Title: title, Description: description}, nil
}
```

This has barely saved any lines of code, but that's rarely the point of refactoring. What I'm trying to do here is just separating the *what* from the *how* of reading lines to make the code a little more declarative to the reader.

Whilst the magic numbers of 7 and 13 get the job done, they're not awfully descriptive.

```go
const (
        titleSeparator       = "Title: "
        descriptionSeparator = "Description: "
)

func newPost(postFile io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postFile)

        readLine := func() string {
                scanner.Scan()
                return scanner.Text()
        }

        title := readLine()[len(titleSeparator):]
```

```go
	description := readLine()[len(descriptionSeparator):]
	return Post{Title: title, Description: description}, nil
}
```

Now that I'm staring at the code with my creative refactoring mind, I'd like to try making our readLine function take care of removing the tag. There's also a more readable way of trimming a prefix from a string with the function `strings.TrimPrefix`.

```go
func newPost(postBody io.Reader) (Post, error) {
	scanner := bufio.NewScanner(postBody)

	readMetaLine := func(tagName string) string {
		scanner.Scan()
		return strings.TrimPrefix(scanner.Text(), tagName)
	}

	return Post{
		Title:       readMetaLine(titleSeparator),
		Description: readMetaLine(descriptionSeparator),
	}, nil
}
```

You may or may not like this idea, but I do. The point is in the refactoring state we are free to play with the internal details, and you can keep running your tests to check things still behave correctly. We can always go back to previous states if we're not happy. The TDD approach gives us this license to frequently experiment with ideas, so we have more shots at writing great code.

The next requirement is extracting the post's tags. If you're following along, I'd recommend trying to implement it yourself before reading on. You should now have a good, iterative rhythm and feel confident to extract the next line and parse out the data.

For brevity, I will not go through the TDD steps, but here's the test with tags added.

```go
func TestNewBlogPosts(t *testing.T) {
	const (
		firstBody = `Title: Post 1
Description: Description 1
Tags: tdd, go`
		secondBody = `Title: Post 2
Description: Description 2
Tags: rust, borrow-checker`
        )

    // rest of test code cut for brevity
    assertPost(t, posts[0], blogposts.Post{
        Title:       "Post 1",
        Description: "Description 1",
        Tags:        []string{"tdd", "go"},
    })
}
```

You're only cheating yourself if you just copy and paste what I write. To make sure we're all on the same page, here's my code which includes extracting the tags.

```
const (
        titleSeparator       = "Title: "
        descriptionSeparator = "Description: "
        tagsSeparator        = "Tags: "
)

func newPost(postBody io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postBody)

        readMetaLine := func(tagName string) string {
                scanner.Scan()
                return strings.TrimPrefix(scanner.Text(), tagName)
        }

        return Post{
                Title:       readMetaLine(titleSeparator),
                Description: readMetaLine(descriptionSeparator),
                Tags:        strings.Split(readMetaLine(tagsSeparator), ", "),
        }, nil
}
```

Hopefully no surprises here. We were able to re-use `readMetaLine` to get the next line for the tags and then split them up using `strings.Split`.

The last iteration on our happy path is to extract the body.

Here's a reminder of the proposed file format.

```
Title: Hello, TDD world!
Description: First post on our wonderful blog
Tags: tdd, go
---
Hello world!

The body of posts starts after the `---`
```

We've read the first 3 lines already. We then need to read one more line, discard it and then the remainder of the file contains the post's body.

Write the test first

Change the test data to have the separator, and a body with a few newlines to check we grab all the content.

```
        const (
                firstBody = `Title: Post 1
Description: Description 1
Tags: tdd, go
---
Hello
World`
                secondBody = `Title: Post 2
Description: Description 2
Tags: rust, borrow-checker
---
B
L
M`
    )
```

Add to our assertion like the others

```
        assertPost(t, posts[0], blogposts.Post{
        Title:       "Post 1",
        Description: "Description 1",
        Tags:        []string{"tdd", "go"},
        Body: `Hello
World`,
    })
```

Try to run the test

```
./blogpost_test.go:60:3: unknown field 'Body' in struct literal of type blogposts.Pos
```

As we'd expect.

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

Add `Body` to `Post` and the test should fail.

```
=== RUN   TestNewBlogPosts
    blogposts_test.go:38: got {Title:Post 1 Description:Description 1 Tags:[tdd go] B
        World}
```

Write enough code to make it pass

1. Scan the next line to ignore the `---` separator.
2. Keep scanning until there's nothing left to scan.

```
func newPost(postBody io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postBody)

        readMetaLine := func(tagName string) string {
                scanner.Scan()
                return strings.TrimPrefix(scanner.Text(), tagName)
        }

        title := readMetaLine(titleSeparator)
        description := readMetaLine(descriptionSeparator)
        tags := strings.Split(readMetaLine(tagsSeparator), ", ")

        scanner.Scan() // ignore a line

        buf := bytes.Buffer{}
        for scanner.Scan() {
                fmt.Fprintln(&buf, scanner.Text())
        }
        body := strings.TrimSuffix(buf.String(), "\n")

        return Post{
                Title:       title,
                Description: description,
                Tags:        tags,
                Body:        body,
        }, nil
}
```

- `scanner.Scan()` returns a `bool` which indicates whether there's more data to scan, so we can use that with a `for` loop to keep reading through the data until the end.
- After every `Scan()` we write the data into the buffer using `fmt.Fprintln`. We use the version that adds a newline because the scanner removes the newlines from each line, but we need to maintain them.
- Because of the above, we need to trim the final newline, so we don't have a trailing one.

Refactor

Encapsulating the idea of getting the rest of the data into a function will help future readers quickly understand *what* is happening in `newPost`, without having to concern themselves with implementation specifics.

```go
func newPost(postBody io.Reader) (Post, error) {
        scanner := bufio.NewScanner(postBody)

        readMetaLine := func(tagName string) string {
                scanner.Scan()
                return strings.TrimPrefix(scanner.Text(), tagName)
        }

        return Post{
                Title:       readMetaLine(titleSeparator),
                Description: readMetaLine(descriptionSeparator),
                Tags:        strings.Split(readMetaLine(tagsSeparator), ", "),
                Body:        readBody(scanner),
        }, nil
}

func readBody(scanner *bufio.Scanner) string {
        scanner.Scan() // ignore a line
        buf := bytes.Buffer{}
        for scanner.Scan() {
                fmt.Fprintln(&buf, scanner.Text())
        }
        return strings.TrimSuffix(buf.String(), "\n")
}
```

Iterating further

We've made our "steel thread" of functionality, taking the shortest route to get to our happy path, but clearly there's some distance to go before it is production ready.

We haven't handled:

- when the file's format is not correct
- the file is not a `.md`

- what if the order of the metadata fields is different? Should that be allowed? Should we be able to handle it?

Crucially though, we have working software, and we have defined our interface. The above are just further iterations, more tests to write and drive our behaviour. To support any of the above we shouldn't have to change our *design*, just implementation details.

Keeping focused on the goal means we made the important decisions, and validated them against the desired behaviour, rather than getting bogged down on matters that won't affect the overall design.

Wrapping up

`fs.FS`, and the other changes in Go 1.16 give us some elegant ways of reading data from file systems and testing them simply.

If you wish to try out the code "for real":

- Create a `cmd` folder within the project, add a `main.go` file
- Add the following code

```
import (
    blogposts "github.com/quii/fstest-spike"
    "log"
    "os"
)

func main() {
        posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
        if err != nil {
                log.Fatal(err)
        }
        log.Println(posts)
}
```

- Add some markdown files into a `posts` folder and run the program!

Notice the symmetry between the production code

```
posts, err := blogposts.NewPostsFromFS(os.DirFS("posts"))
```

And the tests

```
posts, err := blogposts.NewPostsFromFS(fs)
```

This is when consumer-driven, top-down TDD *feels correct*.

A user of our package can look at our tests and quickly get up to speed with what it's supposed to do and how to use it. As maintainers, we can be *confident our tests are useful because they're from a consumer's point of view*. We're not testing implementation details or other incidental details, so we can be reasonably confident that our tests will help us, rather than hinder us when refactoring.

By relying on good software engineering practices like **dependency injection** our code is simple to test and re-use.

When you're creating packages, even if they're only internal to your project, prefer a top-down consumer driven approach. This will stop you over-imagining designs and making abstractions you may not even need and will help ensure the tests you write are useful.

The iterative approach kept every step small, and the continuous feedback helped us uncover unclear requirements possibly sooner than with other, more ad-hoc approaches.

Writing?

It's important to note that these new features only have operations for *reading* files. If your work needs to do writing, you'll need to look elsewhere. Remember to keep thinking about what the standard library offers currently, if you're writing data you should probably look into leveraging existing interfaces such as `io.Writer` to keep your code loosely-coupled and re-usable.

Further reading

- This was a light intro to `io/fs`. Ben Congdon has done an excellent write-up which was a lot of help for writing this chapter.
- Discussion on the file system interfaces