Custom Packages: Folder Structure, go install and go test

This lesson covers important concepts like folder structure of a custom package, testing the executable and installation under workspace.

WE'LL COVER THE FOLLOWING

For demonstration, we take a simple package `uc` which has a function `UpperCase` to turn a string into uppercase letters. This is just for demonstration purposes (it wraps the same functionality from package "strings"), but the same techniques can be applied to more complex packages.

Folder-structure for custom packages

The following structure is considered best practice and imposed by the `go tool` (where `uc` stands for a general package name; the names in bold are folders; italicized is the executable):

```
go_projects (a workspace in $GOPATH)
           src/uppercase

                       /uc (contains go code for package uc)
                           uc.go
                           uc_test.go
                       /uc_main
```

```
                        /uc_main
                            ucmain.go (main program for using package uc)

            pkg/windows_amd64 (the actual name depends on your operating sy
stem/architecture)

                        /uppercase
                        uc.a (object file of package)

bin                     (contains the final executable files)

                         uc_main.exe
```

Building the package `uc` in the *uc* folder

The functionality is implemented in uc.go, belonging to the package `uc`:

```
package uc
import "strings"

func UpperCase(str string) string {
  return strings.ToUpper(str)
}
```

From the app's folder (**$GOPATH/src/uppercase**), build and install the package locally with the command:

```
go install uppercase/uc
```

This copies the package `archive uc.a` to **pkg/os_arch/uppercase**.

Testing the uc package: `go test`

Go can test our package `uc`. To do that, we write 1 (or more) test source files whose names end with **_test.go**, and that import the package `testing`. They must contain functions named `TestXXX` with signature `func (t*testing.T)`. The test framework invoked by `go test` runs each such function. If the function calls a failure function such as `t.Error` or `t.Fail`, the test is considered to have failed. Add a test to the `uc` package by creating the file **$GOPATH/src/uppercase/uc/uc_test.go** containing the following Go code:

```
package uc
import "testing"
```

```
type ucTest struct {

  in, out string
}

var ucTests = []ucTest {
ucTest{"abc", "ABC"},
ucTest{"Go", "GO"},
ucTest{"Antwerp", "ANTWERP"},
}

func TestUC(t *testing.T) {
  for _, ut := range ucTests {
    uc := UpperCase(ut.in)
    if uc != ut.out {
      t.Errorf("UpperCase(%s) = %s, must be
      %s.", ut.in, uc, ut.out)
    }
  }
}
```

Go to the package folder **$GOPATH/src/uppercase** and test it with:

```
go test uppercase/uc
```

which produces as output:

```
ok uppercase/uc 0.155s
```

or with more verbosity (-v):

```
go test –v uppercase/uc
```

which produces as output:

```
=== RUN TestUC
--- PASS: TestUC (0.00 seconds
PASS
ok uppercase/uc 0.091s
```

The command `go test ./...` will run all test code from the packages in and beneath the current directory.

Building the executable:

Then we make our main starting program which uses the `uc` package as **ucmain.go** in folder **uppercase/uc_main**:

```
package main
import (
"fmt"
"uppercase/uc"
)

func main() {
  str1 := " USING package uc!"
  fmt.Println(uc.UpperCase(str1))
}
```

Then issue the following command in the package folder:

```
go install uppercase/uc_main
```

which puts the executable **uc_main.exe** in the **bin** folder. Running **uc_main** gives as output:

```
USING PACKAGE UC!
```

If the go command has no path-parameter, it operates on the current directory only.

```
cd /path/to/package
go build # build package to local directory
go install # install package to $GOPATH/bin or $GOBIN
go test # test package in local directory
```

Installing under $GOROOT:

If we want the package *to be used from any Go-program on the system*, it must be installed under **$GOROOT**. To do this, set `GOPATH = $GOROOT` in .profile and .bashrc; then `go install uppercase` will:

- Copy the source code to **$GOROOT/src/pkg/os_arch/uppercase**
- Copy the package archive to **$GOROOT/pkg/os_arch/uppercase**

The package can then be imported in any Go-source as:

```
import uc
```

Alternatives for setting up your Go environment

- Put every project inside its own workspace:
  GOPATH=/path/to/proj1:/path/to/proj2:...
- Separate your own projects from 3rd party projects:

```
/home/user/goprojects
        /own
                /bin
                /pkg
                /src
        /3rdparty
        /bin
                /pkg
                /src
```

This can be accomplished with **GOPATH=$HOME/projects/3rdparty:$HOME/projects/own**.

OS-dependent code

Your program should rarely be written differently according to the operating system on which it is going to run; in the vast majority of cases, the language and standard library handle most portability issues. You could have an excellent reason to write platform-specific code such as assembly language support. In that case, it is reasonable to follow this convention:

```
prog1.go
prog1_linux.go
prog1_darwin.go
prog1_windows.go
```

prog1.go defines the common code interface independent from operating systems, put the OS-specific code in its own Go-file named **prog1_os.go**.

That's it about the custom packages. In the next lesson, you'll learn the distribution of Go code.