Go Work Environment

This lesson addresses how compilation looks in the background, what a workspace is and what the basic folder structure of a Go environment is.

WE'LL COVER THE FOLLOWING

Workspace

A folder structure with two directories at its root is typically is called a *workspace*. A typical Golang project keeps all its code in a *single* workspace. The workspace of Golang is:

- *src*, a folder that contains Go source code files
- *bin*, a folder that contains executable binaries (also called commands)

However, if you need external packages, there will also be a folder called *pkg* to accommodate these.

GOPATH variable

The workspace root location *cannot* be the same as the Go install location, and it is specified by the GOPATH environment variable. It is by default a go subfolder in the home directory, so **$HOME/go** on Unix or **C:\Users\username\go** on Windows. Add the *bin* folder of your workspace to the PATH variable so that your application binaries are automatically found. For example on Linux add the following to your .profile:

```
export GOPATH=$HOME/go # this is the default
export PATH=$PATH:$GOPATH/bin
```

On Windows, GOPATH can also be added to the *System Variables*. GOPATH can be any other folder, for example, **e:\go_projects**, but, then you have to set this explicitly. The *src* branch can contain many code repositories, which are version controlled (managed by Git or Mercurial). If you keep your code at GitHub under github.com/user, then each repository folder is named with that base path, like this:

```
- src
 + github.com/user/repo1
 + github.com/user/repo2
 + ...
 + (can be any remote package installed with go get)
```

Each repository contains one or more packages. Each package is a *single subfolder* containing one or more Go code files.

Other Go environment variables

We already encountered these two:

- GOROOT—Go installation folder
- GOPATH—Your application working directory

Cross-compiling

Compiling on a host machine that has other characteristics than the target machine, where you are going to run the Go application, is called **cross-compiling**.

The Go-environment can be further customized by several environment variables. These are mostly *optional* and set by the installation procedure.

However, you can override them when needed. The Go compiler architecture enables *cross-compiling*. By definition, there are *two* machines:

- Target machine
- Host machine

Target machine

GOARCH indicates the processor-architecture of the target machine and can have one of the values from the 3rd column of the figure below, like 386, amd64, arm. **GOOS** indicates the operating system of the target machine and can have one of the values from the 4th column of the figure below, like darwin, Freebsd, Linux, Windows.

Host machine

This is the local system. **GOHOSTOS** and **GOHOSTARCH** are the name of the host operating system and compilation architecture. By default, target and host architecture are the same, and **GOHOSTOS** and **GOHOSTARCH** take their values from **GOOS** and **GOARCH**.

	Host (H) developing on		Target (T) running on	
Examples	**GOHOSTARCH**	**GOHOSTOS**	**GOARCH**	**GOOS**
H: 64 bit Linux T: 32 bit Linux	amd64	linux	386	linux
H: 32 bit Linux T: 64 bit Linux	386	linux	amd64	linux
H: 32 bit Windows T: 64 bit Linux	386	windows	amd64	linux

Cross Compiling to other Platforms/Architecture

The **GOMAXPROCS** variable can also be of use. This specifies the number of cores or processors your application uses, as discussed in Chapter 12.

On a Unix system, set these variables (if needed) in your shell profile, which is the $HOME/.profile or $HOME/.bashrc file or equivalent (use any editor like gedit or vi) as:

```
export GOARCH=amd64
export GOOS=linux
```

The **go run** command does not produce an executable file output; it stores its output in a folder specified by **GOCACHE**. This has the default value

$HOME/.cache/go-build on Unix and C:\Users\user\AppData\Local\go-build on a Windows machine.

The command **go env** lists *all* Go environment variables.

Runtime environment

Although the compiler generates native executable code, this code executes within a runtime (the code of this tool is contained in the package runtime). This runtime is minimal compared to the virtual machines used by Java and .NET-languages. It is responsible for handling memory allocation and garbage collection (as explained in Chapter 8), stack handling, goroutines, channels, slices, maps, reflection, and more. Package runtime is the "top-level" package that is linked to every Go package, and it is mostly written in C. It can be found in $GOROOT/src/runtime/. Go executable files are much bigger in size than the source code files. This is precisely because the Go runtime is embedded in every executable. This could be a drawback when having to distribute the executable to a large number of machines. On the other hand, deployment is much easier than with Java or Python, because with Go, everything needed sits in 1 static binary, no other files are needed. There are no dependencies that can be forgotten or incorrectly versioned.

Garbage collector

Go has a very efficient, low-latency concurrent collector, based on the mark-and-sweep algorithm. Having a garbage-collected language doesn't mean you can ignore memory allocation issues like allocating and deallocating memory also uses CPU-resources.

Setting variables of the Go environment accurately provides a perfect workspace to enable successful compilation. The next lesson covers the basic requirements for a Go environment.