THE EXPERT'S VOICE® IN GO

Web Development with Go

Building Scalable Web Apps and RESTful Services

Shiju Varghese

Apress®

Web Development with Go

Building Scalable Web Apps and RESTful Services

Shiju Varghese

Apress®

Web Development with Go

ISBN-13 (pbk): 978-1-4842-1053-6

ISBN-13 (electronic): 978-1-4842-1052-9

Managing Director: Welmoed Spahr
Lead Editor: Celestin John Suresh
Technical Reviewer: Prateek Baheti
Editorial Board: Steve Anglin, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Jill Balzano
Copy Editor: Nancy Sixsmith
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

I would like to dedicate this book to my parents, the late C.S. Varghese and Rosy Varghese. I would like to thank them for their unconditional love and life struggles for the betterment of our lives.

I would like to dedicate this book to my lovely wife Rosmi and beautiful daughter Irene Rose. Without their love and support, this book would not have been possible.

Finally, I would like to dedicate this book to my elder sister Shaijy and younger brother Shinto.

—Shiju Varghese

Contents at a Glance

Contents

About the Author

Shiju Varghese is a solutions architect focused on building highly scalable Cloud native applications with a special interest in APIs, Microservices, containerized applications, and distributed systems. He currently specializes in Go, Google Cloud, and Docker. Shiju is passionate about building scalable back-end systems and Microservices in Go. He is a pragmatic minimalist who focuses on real-world practices for architecting solutions. Shiju worked extensively in C# and Node.js before adopting Go as the primary technology stack. As a consulting solutions architect, he provides guidance and solutions for the successful adoption of Go in enterprises and startups**.**

About the Technical Reviewer

Prateek Baheti is a senior application developer at Thoughtworks, a global software company. He has worked in the test automation space for the past 3 years and has been a major contributor to the open source test-automation tool, Gauge (which is primarily written in Golang).
A practitioner of agile software development, Prateek has experience with building tools and services in Java and Ruby on Rails. He is a polyglot programmer and a tech enthusiast. Prateek loves traveling, going on long drives, and spending quality time at home with his family. You can find him at tech conferences, watching movies, or exploring new restaurants and breweries.

Introduction

Go, often referred to as Golang, is a general-purpose programming language that was developed at Google in November 2009.

Several programming languages are available for writing different kinds of software systems, and some languages have existed for decades. Some mainstream programming languages are evolving by adding new features in their newer versions, which are released with many new features in each version. Both C# and Java provide too many features in their language specification.

At the same time, lots of innovations and evolutions are happening for the computer hardware and IT infrastructure. Software systems are written with feature-rich programming languages, but we can't leverage the power of modern computers and IT infrastructures by using them. We are using programming languages that were created in the era of single-core machines, and now we write applications for multicore machines using these languages.

Just like everything else, computer programming languages are evolving. Go is an evolutionary language for writing software systems for modern computers and IT infrastructures using a simple and pragmatic programming language. On the Go web site at https://golang.org/, Go is defined as follows: "Go is an open source programming language that makes it easy to build simple, reliable, and efficient software."

Go is designed for solving real-world problems instead of academic theories and intellectual thoughts. Go is a pragmatic programming language that ignores the programming language theory (PLT) that has evolved in the last three decades; it provides a simple programming model for building efficient software systems with first-class support for concurrency. Go's built-in Concurrency feature gives you an exciting programming experience for writing highly efficient software systems by leveraging concurrency. For every programming language, there is a design goal. Go is designed to be a simple programming language, and it excels as a simple and pragmatic language.

Go is the language of choice for building many innovative software systems, including Docker, Kubernetes, and others. Like Parse MBaaS by Facebook, many existing systems are re-engineering to Go. I have assisted several organizations to successfully adopt Go, and the adoption process was extremely easy thanks to Go's simplicity and pragmatism. I am sure that you will be excited about Go when you develop real-world software systems.

Go is a general-purpose programming language that can be used to build a variety of software systems, including networked servers, system-level applications, infrastructure tools, DevOps, native mobile applications, graphics, the Internet of Things (IoT), and machine learning. Go can be used for building native mobile applications, and I predict that Go will be a great choice for building native Android applications in the near future.

Go is a great choice of language for building web applications and back-end APIs. I highly recommend Go for building massively scalable back-end RESTful APIs. I predict that Go will be the language of choice in the enterprises for building back-end RESTful APIs, the backbone for building modern business applications in this mobility era.

In this book, I assume that you have knowledge of at least one programming language and have some experience in web programming. If you have prior knowledge of Go, it will help you follow along in this book. If you are completely new to Go, I recommend the following tutorial before you start reading the book: http://tour.golang.org/welcome/1.

When you go through the language fundamentals, I recommend accessing the following section of the Go documentation: https://golang.org/doc/effective_go.html.

The primary focus of this book is web development using the Go programming language. Before diving into web development, the book quickly goes through language fundamentals and concurrency, but doesn't delve too deeply, especially regarding concurrency. You should spend some time exploring concurrency if you want to effectively leverage it for your real-world applications. I recommend the following resource for learning more about concurrency and parallelism: http://blog.golang.org/concurrency-is-not-parallelism.

This book explores various aspects of Go web programming, with a focus on providing practical code. Chapter 9, "Building RESTful Services," can help you to start developing real-world APIs in Go.

I have created a GitHub repository for this book at https://github.com/shijuvar/go-web. The repository provides example code for the book and a few example applications in the near future to help you build real-world web applications.

CHAPTER 1

Getting Started with Go

Everything in this world is evolving, including computers and computer programming languages. Ideas and approaches for building applications are also evolving, based on past experience. Although highly evolved modern computers now have many CPU cores (32, 64, 128 and many more), we still cannot leverage the full power of modern computer hardware by using most of our existing programming languages and tools. Our programs still run slowly, even in high-powered servers with many CPU cores.

For the last decade, many existing programming languages have been evolving with many new features. Language authors have been adding these features based on programming language theory (PLT) and other intellectual thoughts, which make the languages more complex. In today's computing, many people prefer a minimalistic and pragmatic approach for writing applications.

Programming languages are used that excel in specific areas. Some programming languages are great for rapid application development, but would not work well for writing high-performance applications. Other programming languages are very efficient for writing these high-performance applications, but would be difficult for writing applications in a productive manner. It would be great if there were a general-purpose language for developing a variety of applications with a greater level of efficiency, performance, productivity, and faster compilation time. The Go language meets these criteria.

This chapter shows you why Go is a great programming language for solving modern programming challenges. You will learn use cases for adopting Go for your next application.

Introducing Go

Go, also referred to as *Golang*, is a general-purpose programming language, developed by a team at Google and many contributors from the open source community (`http://golang.org/contributors`). The language was announced in November 2009, and the first version was released in December 2012. Go is an open source project that is distributed under a BSD-style license. The official web site of the Go project is available at `http://golang.org/`. It is a statically typed, natively compiled, garbage-collected, concurrent programming language that mostly belongs to the C family of languages in terms of basic syntax. Let's look at some of the features of Go to understand its design principles.

Minimalistic Language with Pragmatic Design

The Go programming language can be simply described in three words: simple, minimal, and pragmatic. If you look deeply into the language design of Go, you see its simple and minimalistic approach, coupled with a pragmatic design. You can observe this simplicity with all the Go language features, including the type system. Today, many programming languages provide too many features that make applications more complex for developers. The design goal of Go is to be a simple and minimal language that provides all the necessary features for developing efficient software systems.

Although Go has fewer language features, productivity is not affected by its pragmatic design. A new Go programmer can quickly learn the language and can easily start to develop production-quality applications. Go has simply ignored many language features from the last three decades and focuses on real-world practices instead of academics and programming language theory (PLT).

From a practical perspective, you might say that Go is an object-oriented programming (OOP) language. But Go's object-oriented approach is different from programming languages such as C++, Java, and C#. Go is not a full-fledged OOP language from an academic perspective. Unlike many existing OOP languages, Go does not support inheritance and does not even have a `class` keyword. It uses composition over inheritance through its simple type system. Go's interface type design shows its uniqueness when compared with other object-oriented programming languages.

Is Go an OOP language? The answer is both yes and no. Go language includes all batteries required for writing applications with an object-oriented approach, but it is not a complete OOP language because it lacks some traditional OOP features.

■ **Note** Programming language theory (PLT) is a branch of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages and their individual features.

A Static Type Language with High Productivity

Go is a statically typed programming language, with its syntax loosely derived from the C language. Like C and C++, it compiles natively to machine code, so Just-In-Time (JIT) compilation is not needed to run its programs. (Programming languages such as Java and C# use JIT compilation to run applications.)

For writing applications, a dynamically typed language provides lots of productivity and expressiveness because you don't have to worry about the data types of the variables you use. In dynamically typed languages, the type of expression is known only at runtime, which provides a greater level of productivity and expressiveness in the syntax to quickly build applications, especially web applications. But when working with a dynamic type language, the performance and maintainability of the applications are affected. Sometimes the debugging experience of an application written in a dynamic type language can be very difficult due to its lack of type safety. Even today, developers use static type languages to generate code for their dynamic type languages. For example, JavaScript developers use statically typed languages such as Microsoft TypeScript for type safety, which finally compiles to JavaScript code.

Although static type languages can provide type safety and performance, working with them can affect the productivity of application development, and compiling larger programs can take a long time. It would be great to have a language that provides the power of both static type and dynamic type language to blend the performance and type safety of a static type language with the productivity of a dynamic type language.

Go is that perfect blend of the power of static type languages and the productivity of dynamic type languages. Go can be called a modern C language that provides faster compilation than C, coupled with the productivity of a dynamic type language.

Concurrency Is a Built-In Feature at the Language Level

Computer hardware has evolved to have many CPU cores and more power, but the power of modern computers cannot be leveraged by using the current programming languages and tools. When production applications are run on high-powered servers, there are performance problems, even though CPU utilization is very low. In some programming environments, concurrency and parallelism are available for better efficiency and performance, but these features are available as a separate library and framework, not as a built-in feature at the language level, which adds more complexity when you write concurrent applications.

In Go, concurrency is built into the language and is designed for writing high-performance concurrent applications for modern computers. Concurrency is one of the unique features of the Go language and it is considered a major selling point. Go's concurrency is implemented using two unique features: goroutines and channels. A *goroutine* is a function that can run concurrently with other goroutines. It is a lightweight thread of execution in which many goroutines execute in a single thread that enables more program performance and efficiency. The most important feature of goroutine is that it is managed and executed by Go runtime. Many programming languages provide support for writing concurrent programs, but they are limited only to communication and synchronization among the threads being executed. And most of the existing languages provide support for concurrency through a framework, but not a built-in feature in the language, so it makes restrictions when concurrency is implemented with these languages.

Go provides *channels* that enable communication between goroutines and the synchronization of their executions. With channels, you can send data from one goroutine to another. Channels also provide a greater level of synchronization between goroutines and ensure that two goroutines are running in a known state. Concurrency is a major reason for adopting Go as the language for building highly efficient software systems with greater levels of performance.

Go Compiles Programs Quickly

One of the challenges of writing C and C++ applications is the time needed for compiling programs, which is very painful for developers when they work on larger C and C++ applications. Go is a language designed for solving the programming challenges of existing programming environments. Its compiler is very efficient for compiling programs quickly; a large Go application can be compiled in few seconds, which is attractive to many C and C++ developers who switch to the Go programming environment.

Go as a General-Purpose Language

Different programming languages are used to develop different kinds of applications. C and C++ have been widely used for systems programming and for systems in which performance is very critical. At the same time, working with C and C++ affect the productivity of application development. Some other programming languages, such as Ruby and Python, offer rapid application development that enables better productivity. Although the server-side JavaScript platform `Node.js` is good for building lightweight JSON APIs and real-time applications, it gets a fail when CPU-intensive programming tasks are executed. Another set of programming languages is used for building native mobile applications. Programming languages such as Objective C and Swift are restricted for use only with mobile application development. Various programming languages are used for a variety of use cases, such as systems programming, distributed computing, web application development, enterprise applications, and mobile application development.

The greatest practical benefit of using Go is that it can be used to build a variety of applications, including systems that require high performance, and also for rapid application development scenarios. Although Go was initially designed as a systems programming language, it is also used for developing enterprise business applications and powerful back-end servers. Go provides high performance while keeping high productivity for application development, thanks to its minimalistic and pragmatic design. The Go ecosystem (which includes Go tooling, the Go standard library, and the Go third-party library) provides essential tools and libraries for building a variety of Go applications. The Go Mobile project adds support for building native mobile applications for both Android and iOS platforms, enabling more opportunities with Go.

In the era of cloud computing, Go is a modern programming language that can be used to build system-level applications; distributed applications; networking programs; games; web apps; RESTful services; back-end servers; native mobile applications; and cloud-optimized, next-generation applications. Go is the choice of many revolutionary innovative systems such as Docker and Kubernetes. A majority of tools on the software containerization ecosystem are being written in Go.

■ **Note** Docker is a revolutionary software container platform, and Kubernetes is a container cluster manager. Both are written in Go.

Go Ecosystem

Go is not just a simple programming language; it is also an ecosystem that provides essential tools and features for writing a variety of efficient software systems. The Go ecosystem contains the following components:

- Go language
- Go libraries
- Go tooling

Go language provides essential syntax and features that allows you to write your programs. These programs leverage libraries for reusable pieces of functionality and tooling for formatting code, compiling code, running tests, installing programs, and creating documentations.

Libraries play a key role in the Go ecosystem because Go is designed to be a modular programming language for writing highly maintainable and composable applications. Libraries provide reusable pieces of functionality distributed as *packages*. You can use packages in Go to write software components in a modular and reusable manner to be shared across Go programs, and can easily maintain your applications. The design philosophy of a Go application is to write small pieces of software components through packages and then compose Go applications with these smaller packages. Libraries are available from the standard library and third-party libraries. When you install Go packages from the standard library, they are installed into the Go installation directory. When you install Go, the environment variable `GOROOT` will be automatically added to your system for specifying the Go installation directory. The standard library includes a larger set of packages that provide a wide range of functionality for writing real-world applications. For example, `"net/http"`, a package from the standard library, can be used to write powerful web application and RESTful services.

■ **Note** For documentation about packages from the standard library, go to `http://golang.org/pkg/`.

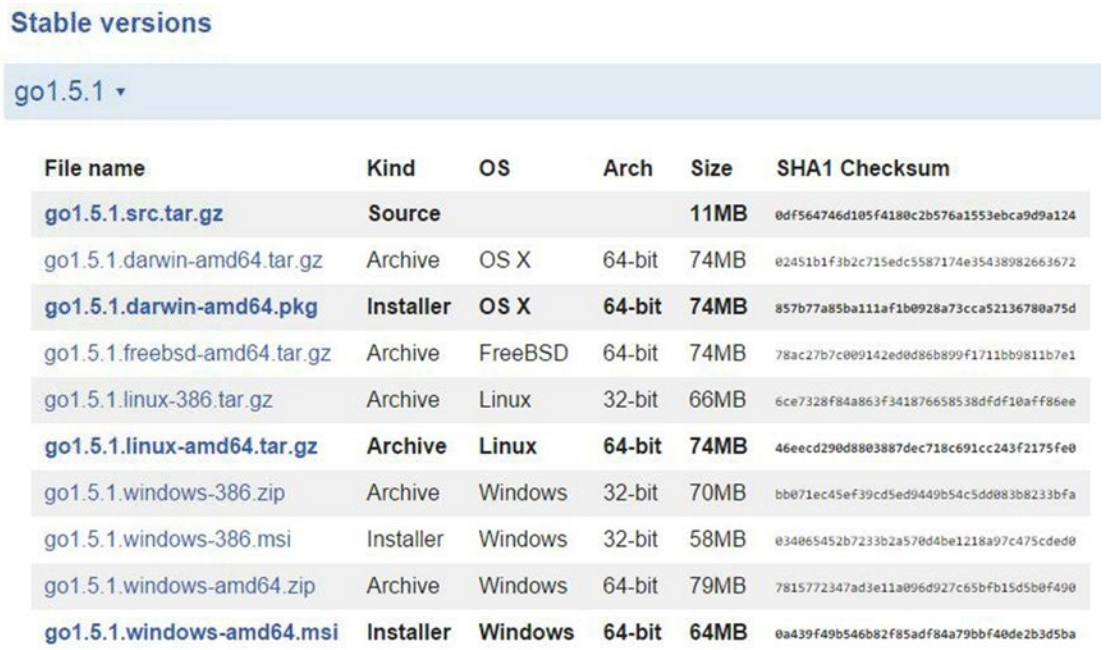
If you need extra functionality not available from the Go standard library, you can leverage third-party libraries provided by the Go developer community, which is very enthusiastic about developing and providing many useful third-party Go packages. For example, if you want to work with the MongoDB database, you can leverage a third-party package called `"mgo"`.

Go tooling is an important component in the Go ecosystem, which provides a number of tooling-support services: building, testing, and installing Go programs; formatting Go code; creating documentation; fetching and installing Go packages; and so on.

Installing the Go Tools

It is easy to install Go on your computers. It provides binary distributions for the FreeBSD, Linux, Mac OS X, and Windows operating systems (OSs); and the 32-bit (386) and 64-bit (amd64) x86 processor architectures. The binary distributions are available at `http://golang.org/dl/`. (If a binary distribution is not available for your combination of OS and architecture, you can install Go from the source.)

Figure 1-1 shows the installer packages and archived sources for Mac, Windows, and Linux platforms, which are listed on the download page of the Go web site. Go provides installers for both Mac and Windows OSs. A package installer is available for Mac X OS that installs the Go distribution to `/usr/local/go` and puts the `/usr/local/go/bin` directory in the `PATH` environment variable.



Stable versions

go1.5.1 ▾

File name	Kind	OS	Arch	Size	SHA1 Checksum
go1.5.1.src.tar.gz	**Source**			**11MB**	0df564746d105f4180c2b576a1553ebca9d9a124
go1.5.1.darwin-amd64.tar.gz	Archive	OS X	64-bit	74MB	02451b1f3b2c715edc5587174e35438982663672
go1.5.1.darwin-amd64.pkg	**Installer**	**OS X**	**64-bit**	**74MB**	857b77a85ba111af1b0928a73cca52136780a75d
go1.5.1.freebsd-amd64.tar.gz	Archive	FreeBSD	64-bit	74MB	78ac27b7c009142ed0d86b899f1711bb9811b7e1
go1.5.1.linux-386.tar.gz	Archive	Linux	32-bit	66MB	6ce7328f84a863f341876658538dfdf10aff86ee
go1.5.1.linux-amd64.tar.gz	**Archive**	**Linux**	**64-bit**	**74MB**	46eecd290d8803887dec718c691cc243f2175fe0
go1.5.1.windows-386.zip	Archive	Windows	32-bit	70MB	bb071ec45ef39cd5ed9449b54c5dd083b8233bfa
go1.5.1.windows-386.msi	Installer	Windows	32-bit	58MB	034065452b7233b2a570d4be1218a97c475cded0
go1.5.1.windows-amd64.zip	Archive	Windows	64-bit	79MB	7815772347ad3e11a096d927c65bfb15d5b0f490
go1.5.1.windows-amd64.msi	**Installer**	**Windows**	**64-bit**	**64MB**	0a439f49b546b82f85adf84a79bbf40de2b3d5ba

Figure 1-1. *Go binary distributions for multiple OSs*

In Mac OS, you can also install Go using Homebrew (`http://brew.sh/`). The following command installs Go on a Mac OS:

```
brew install go
```

An MSI installer is available for the Windows OS that installs Go distribution in `c:\Go`. The installer also puts the `c:\Go\bin` directory in the `PATH` environment variable. Figure 1-2 shows the package installer running for Mac OS.

Figure 1-2. *Go installer for Mac OS*

As mentioned earlier, a successful installation of Go sets up the `GOROOT` environment variable with the location in which the Go tools are installed: `/usr/local/go` (or `c:\Go under Windows`).

You can also install Go tools in a custom location. When you do so, manually configure the environment variable `GOROOT` with the location in which you installed the Go tools on your system.

■ **Note** The complete instructions for downloading and installing Go tools are available at `http://golang.org/doc/install`.

Checking the Installation

You can test the Go installation by typing some Go commands in the terminal window. To verify the installation of Go tools, open the terminal and type the following command:

```
go version
```

Here is the result that shows in a Mac X system:

```
go version go1.4.1 darwin/amd64
```

Here is the result that shows in a Windows system:

```
go version go1.4.1 windows/amd64
```

The following command provides help for Go tools:

```
go help
```

Setting up a Work Environment

Go follows some conventions that organize code in a specific way and help to compile, install, and share Go code more easily. This section discusses how to organize Go code as packages in a workspace.

Go Workspace

Go programs must be kept in a directory hierarchy called a *workspace*, which is simply a root directory of the Go programs.

A workspace contains three subdirectories at its root:

- `src`: This directory contains Go source files organized into packages.
- `pkg`: This directory contains Go package objects.
- `bin`: This directory contains executable commands (executable programs).

When you start working with Go, the initial step is to set up a workspace in which Go programs reside. You must create a directory with three subdirectories for setting up the Go workspace. A Go developer writes Go programs as packages into the `src` directory. Go source files are organized into directories called packages, in which a single directory is used for a single package. You can write two types of packages in Go:

- Packages resulting in executable programs
- Packages resulting in a shared library

The Go tool builds Go packages and installs the resulting binaries into the `pkg` directory if it is a shared library, and into the `bin` directory if it is an executable program. So the `pkg` and `bin` directories are used for storing the output of the packages based on the package type. Keep in mind that you can have multiple workspaces for your Go programs (Go developers typically use a single workspace for their Go programs).

GOPATH Environment Variable

You write Go programs in the workspace, which you should manually specify so that Go runtime knows the workspace location. You can set the workspace location by using the `GOPATH` environment variable. To get started working with Go, create a workspace and set the `GOPATH` environment variable.

Code Organization Paths

You write Go programs as packages into the `GOPATH src` directory. A single directory is used for a single package. Go is designed to easily work with remote repositories such as GitHub and Google Code. When you maintain your programs in a remote source repository, use the root of that source repository as your base path.

For example, if you have a GitHub account at `github.com/user`, it should be your base path. Let's say you write a package named `"mypackage"` at `github.com/user`; your code organization path will be at `%GOPATH%/src/github.com/user/mypackage`. When you import this package to other programs, the path for importing the package will be `github.com/user/mypackage`. If you maintain the source in your local system, you can directly write programs under the `GOPATH src` directory. Suppose that you write a package named `mypackage` on a local system; your code organization path will be at `%GOPATH%/src/mypackage`, and the path for importing the package will be `mypackage`.

Writing Go Programs

Once you create a workspace and set the `GOPATH` environment variable, you can start working with Go. Let's write few simple programs in Go to get started.

Writing a Hello World Program

Let's start by writing a Hello World program, as shown in Listing 1-1.

Listing 1-1. Hello World Program in Go

```
1 package main
2 import "fmt"
3 func main() {
4       fmt.Println("Hello, world")
5 }
```

Line 1: Go programs are organized as packages, and the package name here is specified as `main`. If you name a package `main`, it has a special meaning in Go: the resulting binary will be an executable program.

Line 2: The `"fmt"` package, which provides the functionality for format and print data, is imported from the standard library. The keyword `import` is used for importing packages.

Line 3: The keyword `func` is used to define a function. The function `main` will be the entry point of an executable program and will be executed when the application runs. The package `main` will have one `main` function.

Line 4 The function `Println` is provided by the package `fmt to` print the data. Note that the name of the `Println` function started with an uppercase letter. In Go, all identifiers that start with an uppercase letter are exported to other packages so they will be available to call in other packages.

Let's compile and run the sample program using the Go tool. Navigate to the package directory and then type the `run` command to run the program. Suppose that the location of the package directory is at `github.com/user/hello`:

```
cd $GOPATH/src/github.com/user/hello
go run main.go
```

The preceding command simply prints the phrase "Hello, world". The `run` command compiles the source and runs the program. You can also use the `build` and `install` commands with the Go tool to build and install Go programs that produce binary executables to be run later.

The `build` command compiles the package and puts the resulting binary into the package folder:

```
cd $GOPATH/src/github.com/user/hello
go build
```

The name of the resulting binary is the same as the directory name. If you write this program in a directory named `hello`, the resulting binary will be `hello` (or `hello.exe` under Windows). After compiling the source with the `build` command, you can run the program by typing the binary name.

The `install` command compiles the package and installs the resulting binary into the `bin` directory of `GOPATH`:

```
cd $GOPATH/src/github.com/user/hello
go install
```

You can run this command from any location on your system:

```
go install github.com/user/hello
```

The name of the resulting binary is the same as the directory name. You can now run the program by typing the binary from the `bin` directory of `GOPATH`:

```
$GOPATH/bin/hello
```

If you have added `$GOPATH/bin` to your `PATH` environment variable, just type the binary name from any location on your system:

```
hello
```

Writing a Library

In Go, you can write two types of programs: executable programs and reusable libraries. The previous sample program was an executable program. Let's write a shared library to provide a reusable piece of code to other programs. Create a package directory at the location `$GOPATH/src/github.com/shijuvar/go-web-book/chapter1/calc`.

Listing 1-2 shows a simple package that provides the functionality for adding and subtracting two values.

Listing 1-2. Shared Library Program in Go

```
1 package calc
2
3 func Add(x, y int) int {
4       return x + y
5 }
6 func Subtract(x, y int) int {
7       return x - y
8 }
```

Line 1: The package name is specified as `calc`. The name of the package and package directory must be same.

Line 3: A function named `Add` is defined with the keyword `func`. The name of this function starts with an uppercase letter, so the `Add` function will be exported to other packages. If the name of the function starts with a lowercase letter, it is not exported to other packages, and accessibility will be limited to the same package. Unlike programming in C++, Java, and C#, you don't need to use `private` and `public` keywords to specify the accessibility of identifiers. You can see the simplicity of the Go language throughout the language features. The `Add` method takes two integer parameters and returns an integer value.

Line 6: The `Subtract` function is similar to the `Add` function, but subtracts values between two integer types.

Let's build and install the package. Navigate to the package directory in the terminal window and type the following command:

```
go install
```

The `install` command compiles the source and installs the resulting binary into the `pkg` folder of `GOPATH` (see Figure 1-3). In the `pkg` directory, the `calc` package will be installed at the location `github.com/shijuvar/go-web-book/chapter1/calc` under the platform-specific directory.

Figure 1-3. *Install command installing calc package into pkg folder*

The `install` command behaves a bit differently depending on whether you are creating an executable program or a reusable library. When you run `install` for executable programs the resulting binary will be installed into the `bin` directory of `GOPATH` while it will be installed into the `pkg` directory of `GOPATH` for libraries.

Now you can reuse this package from any program residing in the `GOPATH`. Code reusability in Go is very easy with packages. You have created your first library package. Let's reuse the package code from another executable program (see Listing 1-3).

Listing 1-3. Reusing the calc Package in a Go Program

```
1 package main
2
3 import (
4       "fmt"
5       "github.com/shijuvar/go-web-book/chapter1/calc"
6 )
7
8 func main() {
9       var x, y int = 10, 5
10      fmt.Println(calc.Add(x, y))
11      fmt.Println(calc.Subtract(x, y))
12 }
```

Line 1: Create an executable program.

Lines 3 to 6: These lines import the `"fmt"` package from the standard library and the `"calc"` package from your own library. You can use a single `import` statement to import multiple packages. The path for the packages from the standard library uses short paths such as the `"fmt"` package. For your own packages, you must specify the full path when importing the packages. Using the full path for external packages avoids name conflicts among packages, and you can use the same name for multiple external packages in which the package path would be different.

Line 8: The function `main`, entry point of the package `main`.

Line 9: Declaring two variables, `x` and `y`, with the `int` data type. Go uses the `var` keyword to declare variables in which you can declare multiple variables in a single statement. If you assign values to variables along with the variable declarations, you can use a shorter statement:

```
x,y:=10,5
```

When you use Go's shorter statement for declaring variables, you don't need to specify the variable type because the Go compiler can infer the type, based on the value you assign to the variable. Go provides the productivity of a dynamically typed language while keeping it as a statically typed language. The Go compiler can also do type inference with the `var` statement:

```
var x,y=10,5
```

Lines 10 to 11: You call the exported functions of the `calc` package and reuse the functionality provided by the library.

To run the program, type the following command from the program directory:

```
go run main.go
```

Testing Go Code

The Go ecosystem provides all the essential tools for developing Go applications, including the capability for testing Go code without leveraging any external library or tool. The "testing" package from the standard library provides the features for writing automated tests, and Go tooling provides support for running automated tests. When you develop software systems, writing automated tests for application code is an important practice to ensure quality and improve maintainability. If your code is covered by tests, you can fearlessly refactor your application code.

Let's write some tests for the `calc` package created in the previous section. You create a source file with a name ending in `_test.go`, in which you write tests by adding functions starting with `"Test"` and taking one argument of type `*testing.T`.

In the `calc` package directory, create a new source file named `calc_test.go` that contains the code shown in Listing 1-4.

Listing 1-4. Testing the calc Package

```
1 package calc
2
3 import "testing"
4
5 func TestAdd(t *testing.T) {
6       var result int
7       result = Add(15,10)
8       if result!= 25 {
9               t.Error("Expected 25, got ", result)
10      }
11 }
12 func TestSubtract(t *testing.T) {
13      var result int
14      result = Subtract(15,10)
```

```
15      if result!= 5 {
16              t.Error("Expected 5, got ", result)
17      }
18 }
```

Line 1: Specifies the package name as `calc`.

Line 3: Imports the `"testing"` package from the standard library, which provides the essential functionality for writing tests and works with the `Go test` command.

Line 5: Adds a test named `"TestAdd"` with signature `func (t *testing.T)` for verifying the functionality `Add` function in the `calc` package.

Line 12: Adds a test named `"TestSubtract"` with signature `func (t *testing.T)` for verifying the functionality `Subtract` function in the `calc` package.

To run the tests with the Go tool, type the following command from the package directory:

```
go test
```

The `go test` command identifies and execute tests in the package files, based on the conventions used for testing. It will show the following result:

```
PASS
ok      github.com/shijuvar/go-web-book/chapter1/calc    0.310s
```

Using Go Playground

Go Playground is a tool that allows you to write and run Go programs from your web browser (see Figure 1-4). By using this tool, you can write and run Go programs without having to install Go on your system.

Figure 1-4. *Go Playground*

■ **Note** The browser-based Go Playground tool is available at `https://play.golang.org/`.

Go Playground can also be used to share Go code with other developers. Clicking the Share button provides a sharable URL for sharing your code with others.

Using Go Mobile

You already know that Go can be used as a general-purpose programming language for building a variety of applications. It can also be used for building native mobile applications for both Android and iOS. The Go Mobile project provides tools and libraries for building native mobile applications. It includes a command-line tool called gomobile to help you build these applications.

You can follow two development strategies to include Go into your mobile stack:

- Develop native mobile applications entirely written in Go
- Develop SDK applications by generating bindings from a Go package and invoking them from Java (on Android) and Objective-C (on iOS).

The first strategy is to use Go everywhere in your mobile project by using the packages and tools provided by Go Mobile. Here you can use Go to develop both Android and iOS applications. In the second strategy, you can reuse a Go library package from a mobile application without making significant changes to your existing application. In this strategy, you can share a common code base for both Android and iOS applications. You can write the common functionality once in Go as a library package and put it to the platform-specific code by invoking the Go package through bindings.

■ **Note** You can find out more about the Go Mobile project at `https://github.com/golang/mobile`.

Go as a Language for Web and Microservices

The primary focus of this book is web development using Go. In modern computing, a digital transformation is happening, in which HTTP APIs (often RESTful APIs) are becoming the backbone for web applications, mobile applications, Big Data solutions, and the Internet of Things (IoT). These web-based APIs, which are powering the back ends for many modern applications, enable developers to integrate among various applications.

There has recently been a monolithic architecture approach for developing larger applications, in which a single application includes all the logical components for running the application. These applications are very hard to maintain and scale due to tight coupling among various logical components. To solve various problems found in monolithic application architecture, developers prefer a microservices architectural style, in which a monolithic application is broken into a suite of small services (microservices), each running as an independent unit. The independent service pieces communicate by using either RESTful APIs or message brokers. Go is gradually becoming a preferred language for building RESTful APIs and microservices.

Go may not be the language choice for building traditional web applications in which application logic and UI rendering logic reside in a server-side application. It is, however, the language choice for building modern web applications in which an API, often a RESTful API, is developed as the server-side implementation. By consuming these back-end services, you can build your front-end web applications. A Single Page Application (SPA) architecture has been widely used for building these web applications. The back-end services can also be used for building mobile applications.

Go provides an `HTTP` package that allows you to build high-performance web applications and RESTful services by leveraging the built-in concurrency mechanism provided by Go. In Go, you can quickly build an HTTP server with fewer lines of code and start listening at a given HTTP port. By default, each HTTP request to the web server will be processed concurrently using a goroutine, which is a mechanism in Go to concurrently run functions independently of other functions. You can run millions of goroutines in a single server that enables you to build massively scalable web applications and web APIs in Go.