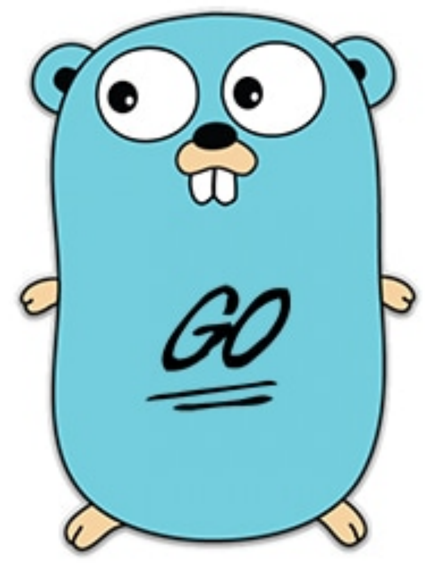
Shiju Varghese's Masterclass

Professional Go

https://shijuvar.medium.com



About Me

- Solutions Architect, focused on Distributed Systems
- Consulting and Training on Go, Distributed Systems, Microservices
- Published Author: *Web Development with Go (Apress, 2015, US edition), Go Recipes (Apress, 2016, US edition)*
- Awarded Microsoft MVP seven times
- Speaker at numerous conferences
- medium.com/@shijuvar | github.com/shijuvar | linkedin.com/in/shijuvar

THE EXPERT'S VOICE® IN GO

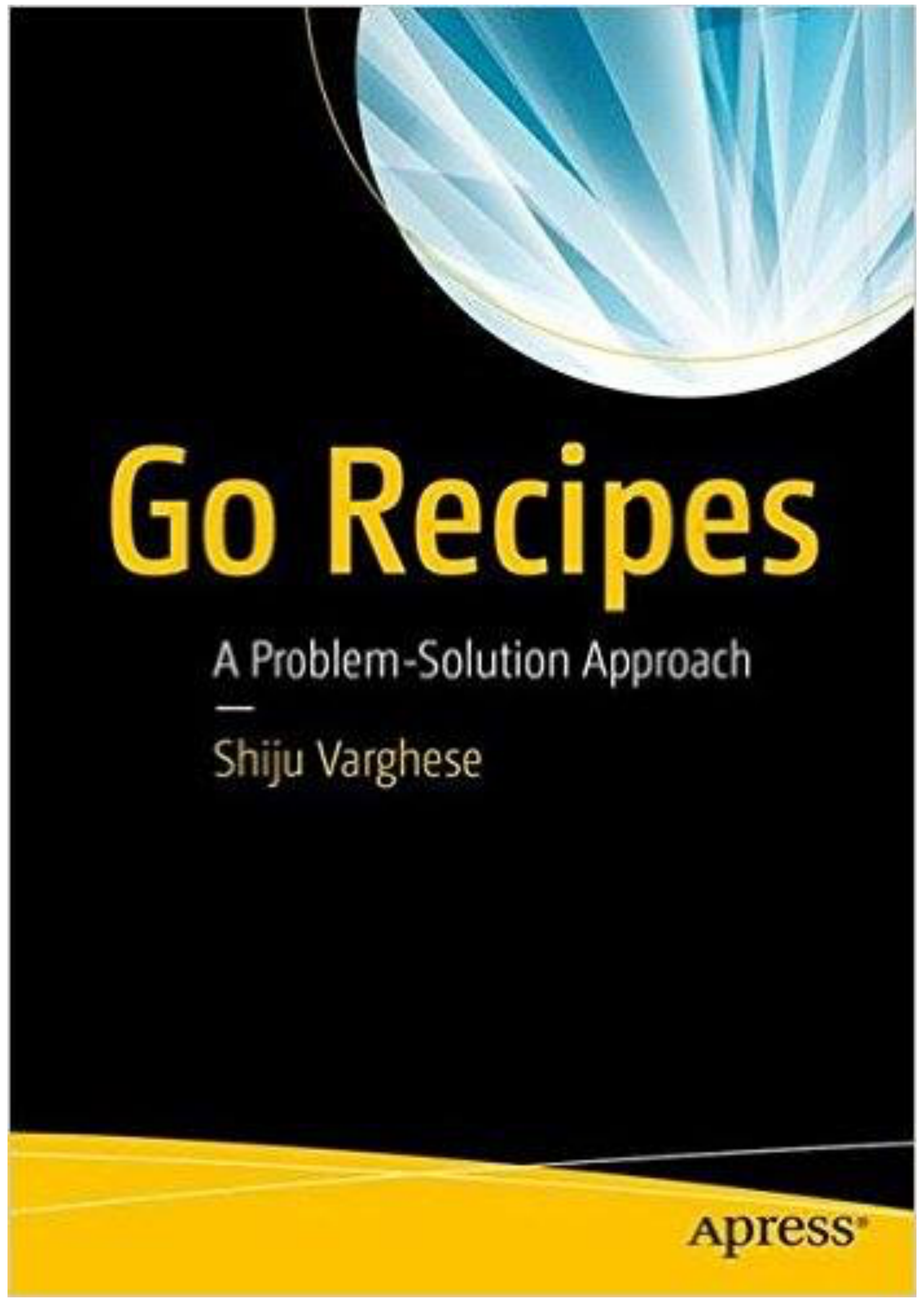
Web Development with Go

Building Scalable Web Apps and RESTful Services

—

Shiju Varghese

Apress®


Go Recipes

A Problem-Solution Approach

—

Shiju Varghese

Apress®

GO

Go

- Go is an open source programming language created at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson
- First appeared in November, 2009
- Go 1.0 was released on March, 2012

Go is an open source programming language that enables the the

production of **simple**, **efficient** and **reliable** software at scale.

The key point here is our programmers are Googlers, they’re not researchers. They’re typically, fairly young, fresh out of school, probably learned Java, maybe learned C or C++, probably learned Python. They’re not capable of understanding a brilliant language but we want to use them to build good software. So, the language that we give them has to be easy for them to understand and easy to adopt.

– Rob Pike, Co-Designer, Go Programming Language

Go’s largest goal is scale

- Creating software at scale
- Running software at scale

Introduction to Go

- An open source programming language created at Google
- Go is simple, minimal and pragmatic
- Modern general-purpose language
- A statically typed language with high productivity
- Compiles to native machine code; Compiles programs quickly
- Garbage-collected language
- Concurrency is a built-in feature

Go Versions

- First appeared in November, 2009
- Go 1.0 was released in March, 2012
- Current version is 1.16

Unlearning is the most important things to master on Go language

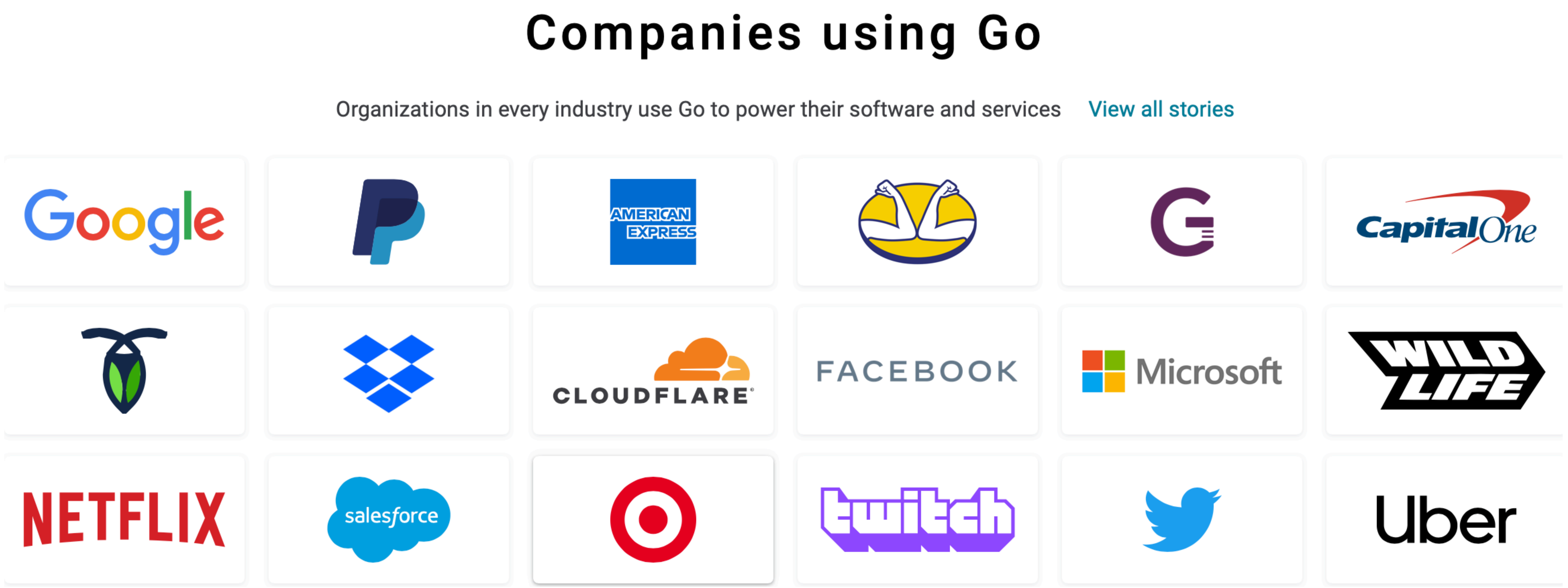
Gopher: An Iconic Mascot of Go

https://blog.golang.org/gopher

Why Go

- Built for writing high performance concurrent programming.
- Modern C for 21st Century.
- Pragmatism and Developer productivity.
- Easier Adoption.
- Go Community.
- Go success stories for building massively scalable applications around the world.
- Language for the era of Cloud, Microservices and Containers.

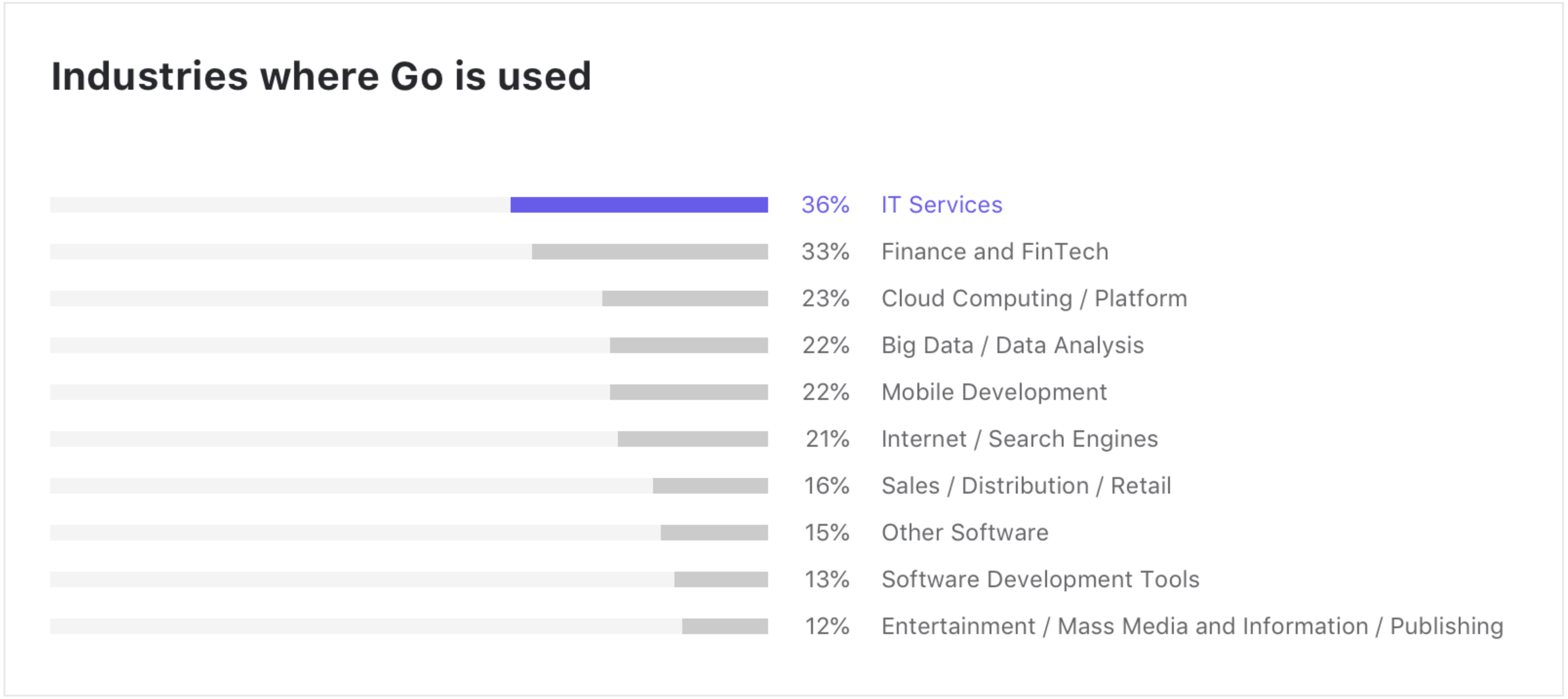
https://go.dev/solutions#case-studies



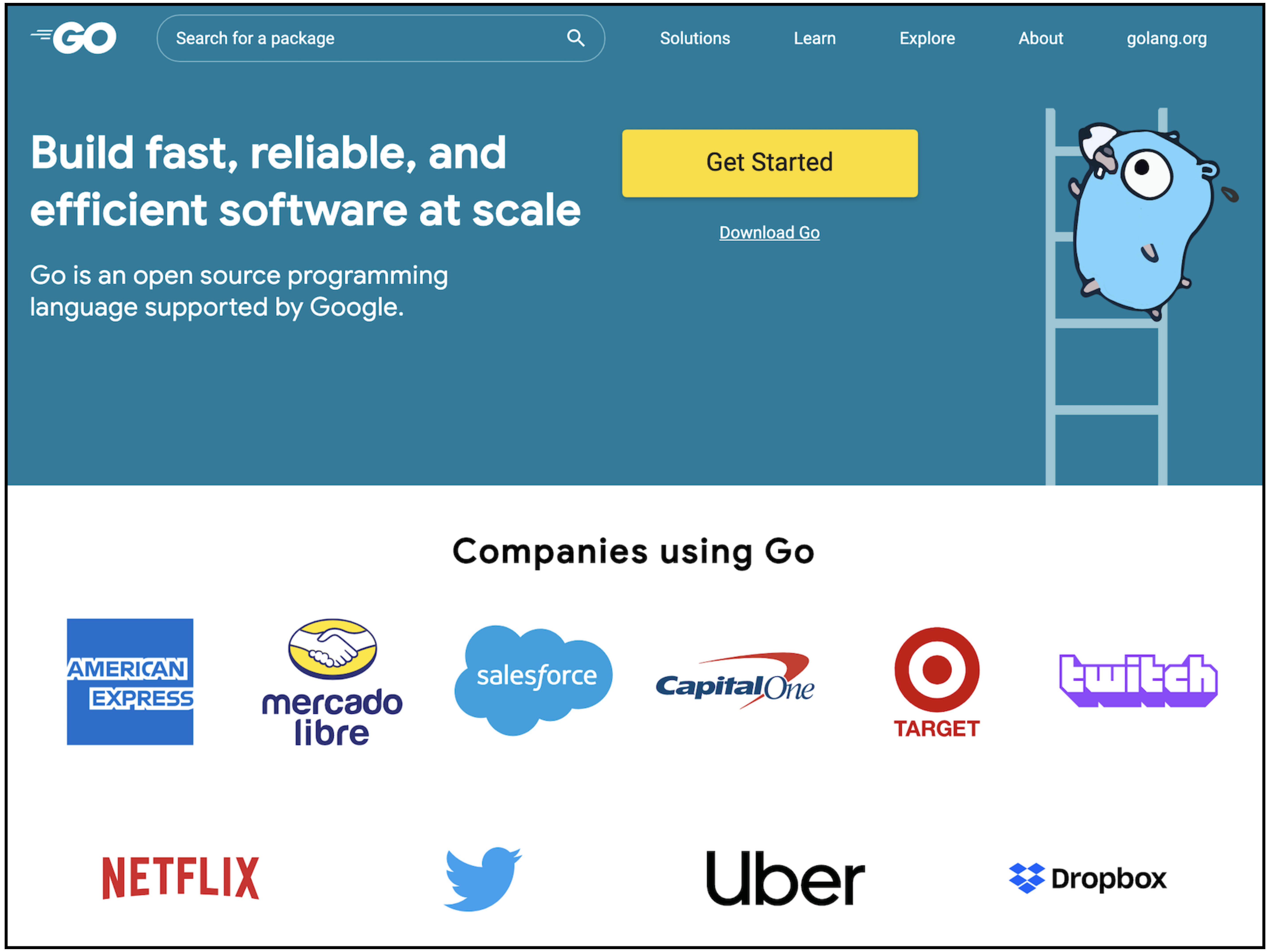
Companies using Go

Organizations in every industry use Go to power their software and services View all stories

Overall, there are about **1.1 million** professional Go developers who use Go as a primary language. But that number is possibly closer to **2.7 million** if we include professional developers who mainly use other programming languages but also do a bit of Go on the side.



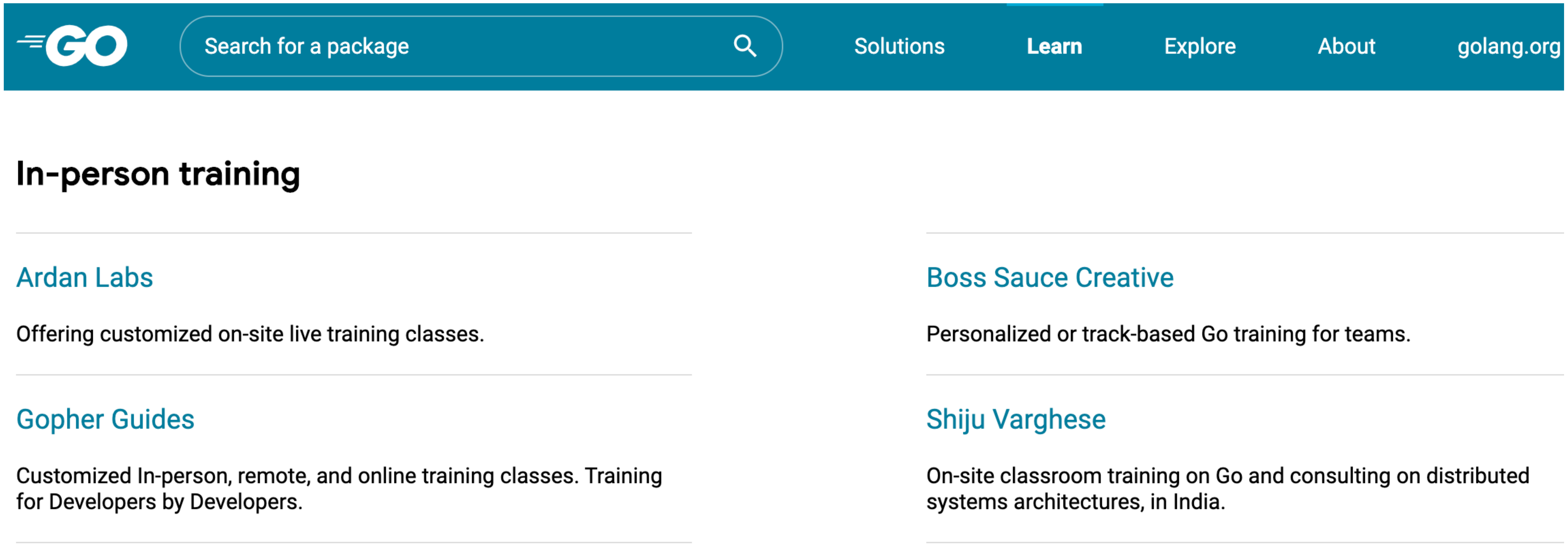
https://go.dev: A hub for Go developers



Build fast, reliable, and efficient software at scale

Go is an open source programming language supported by Google.

Companies using Go



In-person training

Ardan Labs

Offering customized on-site live training classes.

Boss Sauce Creative

Personalized or track-based Go training for teams.

Gopher Guides

Customized In-person, remote, and online training classes. Training for Developers by Developers.

Shiju Varghese

On-site classroom training on Go and consulting on distributed systems architectures, in India.

Hands-on training for writing idiomatic Go code with Clean architecture and SOLID principles

Source Code: https://github.com/shijuvar/gokit (Go basics to architecture references)

Go Ecosystem

- Language
- Tools
- Packages

Go Workspace

A workspace is a directory hierarchy with three directories at its root:

- **src** contains Go source files
- **pkg** contains package objects
- **bin** contains executable commands

Setting up GOPATH

```
export GOPATH=$HOME/go

export PATH=$PATH:$GOPATH/bin
```

Setting up Go development environment in Go 1.16 with Go Modules

Set up Go

- Download and install Go: https://golang.org/dl/
- Set the **GOMODCACHE** environment variable (Eg: export GOMODCACHE=$HOME/gocode/pkg)

GOMODCACHE environment variable

- The module cache is the directory where the go command stores downloaded module files. The module cache is distinct from the build cache, which contains compiled packages and other build artifacts.

- The default location of the module cache is $GOPATH/pkg/mod. To use a different location, set the **GOMODCACHE** environment variable.

Install Directory of Go Programs

- The install directory is controlled by the GOPATH and GOBIN environment variables. If GOBIN is set, binaries are installed to that directory. If GOPATH is set, binaries are installed to the bin subdirectory of the first directory in the GOPATH list. Otherwise, binaries are installed to the bin subdirectory of the default GOPATH ($HOME/go or %USERPROFILE%\go).

Variable Declarations

- The keyword var is used to declare variables.
- Inside the functions, short assignment operator (:=) can be used for declare and initialise variable without using var and datatypes.
- Short assignment operator (:=) can’t be used for declare package level variables.
- The keyword const is used to declare constant variables.

Functions

- The keyword func is used for declare functions.
- Functions can return multiple values.
- Functions can have variadic parameter by using … syntax (eg: …int). The variadic parameter should be the last parameter.
- Go supports anonymous functions.

Functional Go

- Functional programming is based on immutable data structures: No mutable data; No state (or implicit state)
- Functions are first-class citizen in the Go programming language.
- Go functions are like values.
- Go supports Higher-order functions:
    - Takes one or more functions as parameters.
    - Functions can return functions.
- Go supports Closure - A function value that references variables declared outside its body.

Collections

- Go provides three types of data structure to works collections: **array**, **slice** and **map**

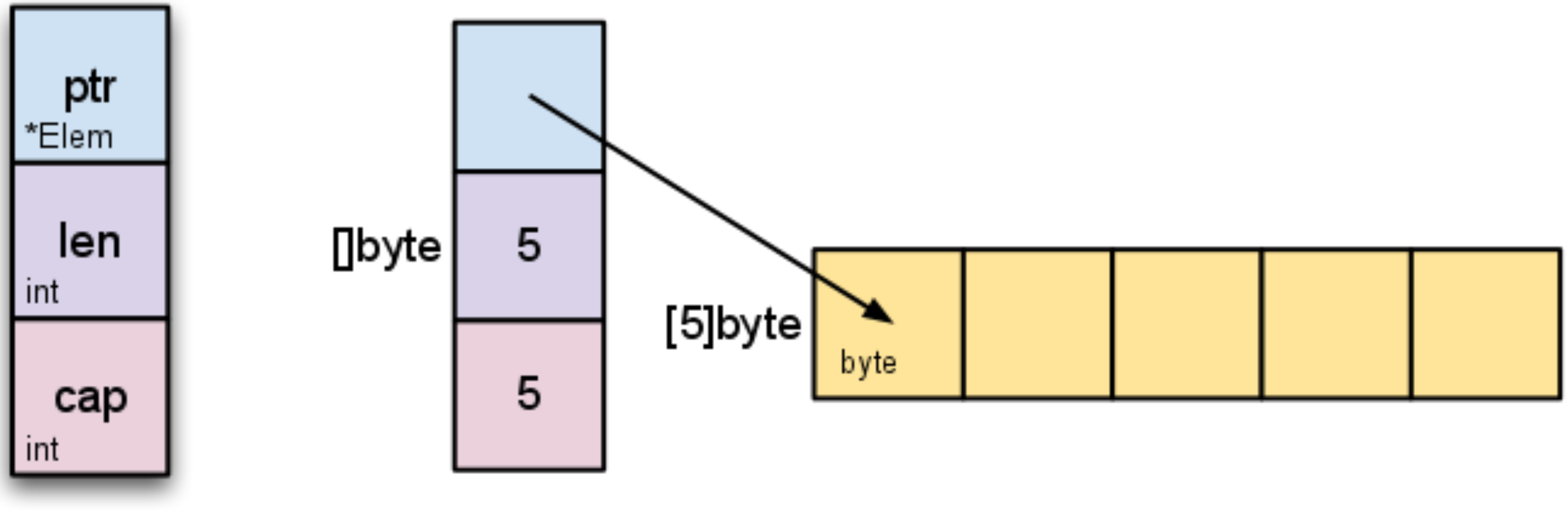
- Arrays are fixed length collection.

- Arrays can be declared and initialised using array literal.

- Slices are dynamically sized collections.

- Slice is a three-field data structure: a pointer points an underlying array, length specifies the length of an underlying array, and capacity specifies the capacity of the array.

Slices



- A slice should be initialised either using make function or slice literal.
- Slices can be enlarged at any time using built-in function append.
- A built-in function copy can be used for creating bigger slices by copying contents from an existing slice.
- Map is a collection of key/value pairs, which is an unordered collection.

Defer, Panic, Recover

- The keyword defer is used to schedule a function call to be executed right after a function returns.

- The built-in panic function is used to stop the execution that will provide a panic situation.

- The built-in recover function is called within a deferred function to regains control over a panicking function.

Packages

- Go programs are organized into directories called packages.
- Go provides two kinds of packages: package main for which the resulting binary will be an executable, and shared library package to be used for other packages.
- The go install command compiles the package and put the resulting binaries into bin subdirectory of GOPATH if it is a package main and put into pkg directory if it is a shared library package.
- Package main should have an entry point: main function.
- Packages identifiers started with uppercase letter will be exported to other packages.
- Packages identifiers started with lowercase letter will not be exported to other packages, but it will be available across the same package.

- Packages can have an init function, which can be used for writing initialising logic at packages level. This will be invoked before invoking main function.
- The keyword import is used to import packages into Go source files.
- Packages can be imported by providing an alias name.
- Packages can be imported by providing _ (underscore) as an alias name, which is used for importing a package just for invoking init functions.
- Third-party packages are installed using go get command.
- The *Go Modules* introduced in Go 1.11, is the official solution to manage dependency management, which may also eliminate the use of GOPATH.

Go Modules & Athens

Challenges with Package Dependencies

- Dependency management of third-party packages
- go get always take latest version of packages

Go Modules

- A module is a collection of related Go packages that are versioned together as a single unit
- A module is defined by a tree of Go source files with a **go.mod** file in the tree's root directory.
- Modules record precise dependency requirements and create reproducible builds
- Modules must be semantically versioned in the form v(major).(minor).(patch), such as v0.1.0, v1.2.3, or v3.0.1 (Check out: https://semver.org)
- Uses minimal version selection (MVS) algorithm

Go Module Behaviour

- Inside GOPATH — defaults to old 1.10 behavior (ignoring modules)
- Outside GOPATH while inside a file tree with a go.mod — defaults to modules behavior
- GO111MODULE environment variable:
  - **unset** or **auto** — default behavior above
  - **on** — force module support on regardless of directory location
  - **off** — force module support off regardless of directory location

Workflow with Go Modules

- ‘**go mod init**’ initialises a module with a file **go.mod**
- Add import statements to .go code as needed (go get is not required)
- 'go get’ will automatically update the go.mod file.
- go get' allows version selection to be overridden by adding an @version suffix or "module query" to the package argument
  - go get github.com/gorilla/mux@v1.6.2
  - go get github.com/gorilla/mux@‘<v1.6.2’

Go Module Commands

- **go mod tidy** - go mod tidy ensures that the go.mod file matches the source code in the module. It adds any missing module requirements necessary to build the current module's packages and dependencies, and it removes requirements on modules that don't provide any relevant packages. It also adds any missing entries to go.sum and removes unnecessary entries.

- go list -m - Print path of main module

- go list -m -f={{.Dir}} - Print root directory of main module

- go list -m all - View final versions that will be used in a build for all direct and indirect dependencies

- go get -u ./... or go get -u=patch ./... (from module root directory) - Update all direct and indirect dependencies to latest minor or patch upgrades

- go build ./... or go test ./... (from module root directory) - Build or test all packages in the module

- go mod vendor - Optional step to create a vendor directory

Minimal Version Selection (MVS)

- The minimal version selection algorithm is used to select the versions of all modules used in a build. For each module in a build, the version selected by minimal version selection is always the semantically highest of the versions explicitly listed by a require directive in the main module or one of its dependencies.

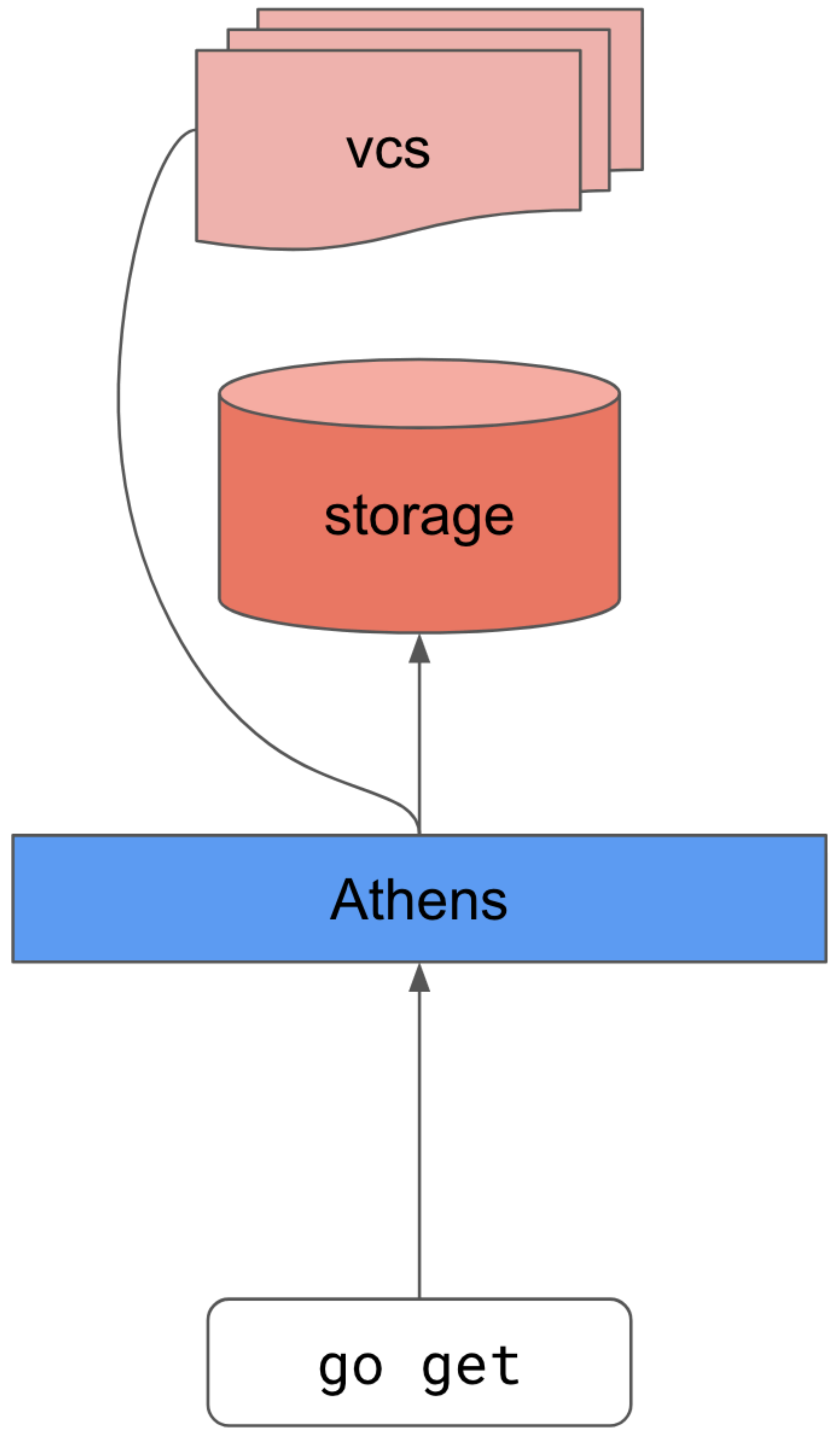
- **MVS Selection:** If your module depends on module A which has a require D v1.0.0, and your module also depends on module B which has a require D v1.1.1, then minimal version selection would choose v1.1.1 of D to include in the build. This selection of v1.1.1 remains consistent even if sometime later a v1.2.0 of D becomes available.

GOMODCACHE environment variable

- The module cache is the directory where the go command stores downloaded module files. The module cache is distinct from the build cache, which contains compiled packages and other build artifacts.
- The default location of the module cache is $GOPATH/pkg/mod. To use a different location, set the **GOMODCACHE** environment variable.

Athens

- GOPROXY Server
- A CDN backed by static storage like a cloud blob store or a filesystem
- A server, backed by a database, that fetches modules from an upstream VCS host as needed



Setting up Athens Proxy

```
$ mkdir -p $(go env GOPATH)/src/github.com/gomods
$ cd $(go env GOPATH)/src/github.com/gomods
$ git clone https://github.com/gomods/athens.git
$ cd athens
$ GO111MODULE=on go run ./cmd/proxy -config_file=./config.dev.toml &
[1] 25243
INFO[0000] Starting application at 127.0.0.1:3000
```

Running Athens Proxy

```
export GOPROXY=http://127.0.0.1:3000
```

User-Defined Type System

- Go provides two types for creating user-defined types: Structs and Interfaces
- Go has no classes
- Methods can be added to any type
- Go has no inheritance
- Interfaces are implicitly satisfied
- Types stand alone by themselves; they just are and have no hierarchy
- Methods aren’t special; they’re just functions

- Go’s type system provides two user defined types: interface, which is used for specifying contracts, and struct, which is used for creating concrete user-defined types

- A struct is a collection of named fields.

- Interface types provide contracts to concrete types, which lets you define behaviours for your objects.

- Behaviours can be added to struct type by specifying method receivers.

- There are two of types of method receivers are available: value receiver and pointer receiver.

- Pointer receivers can be used for two scenarios: Mutates the state of types, and when you deal with type values, which have larger collection of values as fields.

- Go’s type system provides type embedding, which allows you to embedded struct types into other structs to create bigger types using composition.

- **Interfaces are behaviour contracts**

- "In object-oriented programming, a protocol or interface is a common means for unrelated objects to communicate with each other" — Wikipedia

- In Go, explicit declaration of interface implementation is not required. You just need to implement the methods defined in the interface into your struct type to implement an interface type.

- ***interface{}*** (empty interface) says nothing
- ***struct{}*** (empty struct) is a struct type that has no fields
- Empty struct occupies zero bytes of storage

Interface: Design Guidelines

- Provides hidden implementations
- “The bigger the interface, the weaker the abstraction” — Rob Pike
- Return concrete types, receive interfaces as parameters
- “Be conservative in what you send, be liberal in what you accept from others” — Robustness Principle

Struct Methods Vs Free Functions

- If package level variables are used by free functions, then refactor with:
    - Create structs by making package level variables as properties
    - Add methods to the struct
- Use structs for isolating dependencies by making dependencies as properties with interface types. Inject the dependencies from the place where you create instances of those struct types.
- Write free functions where you don't references any state and don't create dependent objects.
- In short, create structs for defining states and dependencies.

Concurrency

Concurrency VS Parallelism

“In programming, concurrency is the *composition* of independently executing processes, while parallelism is the simultaneous *execution* of (possibly related) computations. Concurrency is about *dealing with* lots of things at once. Parallelism is about *doing* lots of things at once.”

Rob Pike, Co-Designer, Go

Concurrency Primitives

- goroutines
- channels

Goroutines

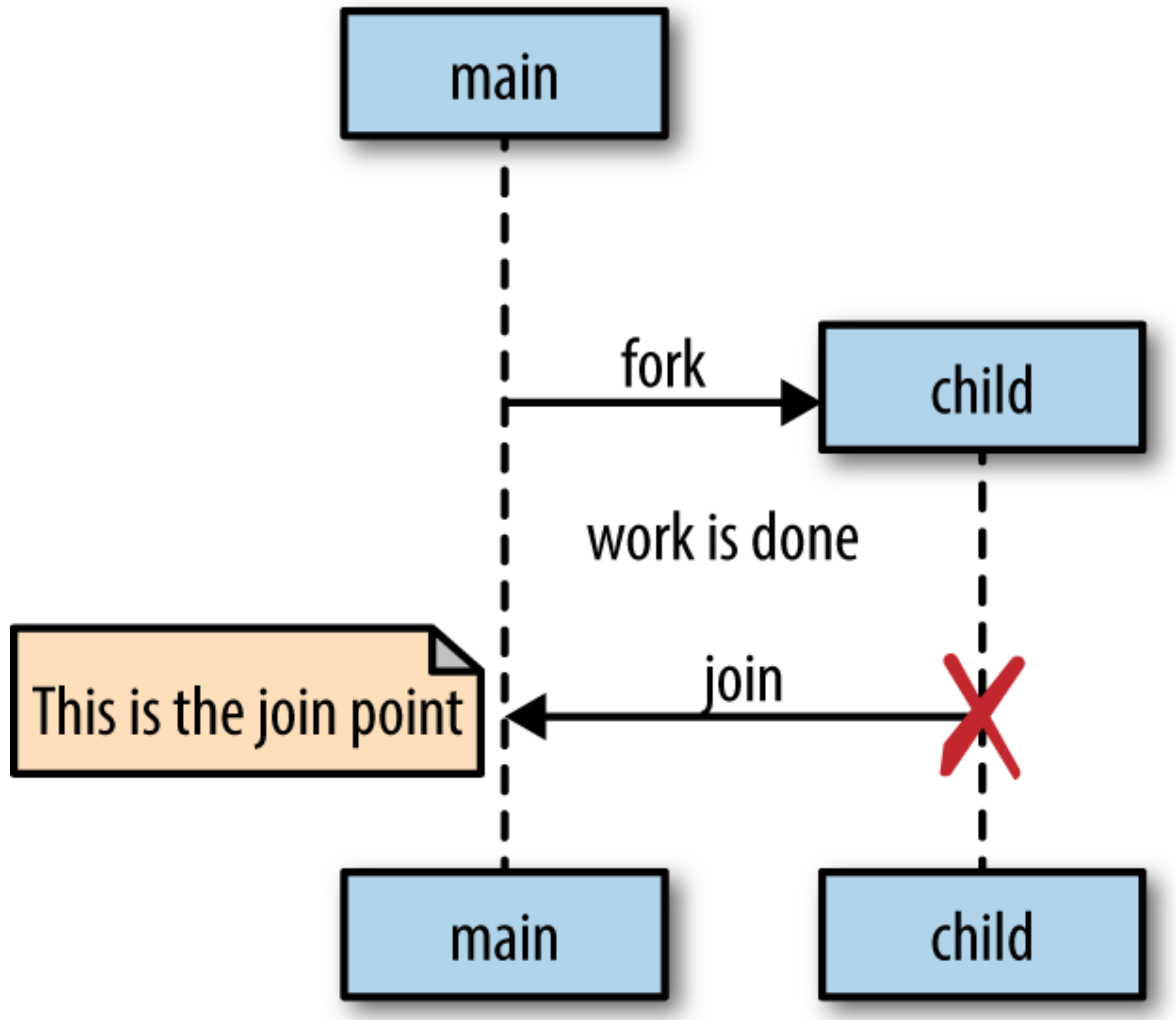
- A goroutine is a function that is running concurrently
- Goroutines can be launched by placing the ***go*** keyword before a function
- Created and managed by Go runtime
- Executing based on a technique called ***m:n Scheduler*** (Maps *m* goroutines (goroutines are kind of **green threads**. A bit more specific word would be "**coroutines**") to *n* OS threads)

Green threads are threads that are scheduled by a runtime library or virtual machine (VM) instead of natively by the underlying operating system.

Go Scheduler

Go scheduler is a ***m:n scheduler***
Multiplexes (or schedules) ***m*** goroutines on ***n*** OS threads.
Scheduler is responsible to schedule the goroutines on limited number of OS threads

Fork-Join Model



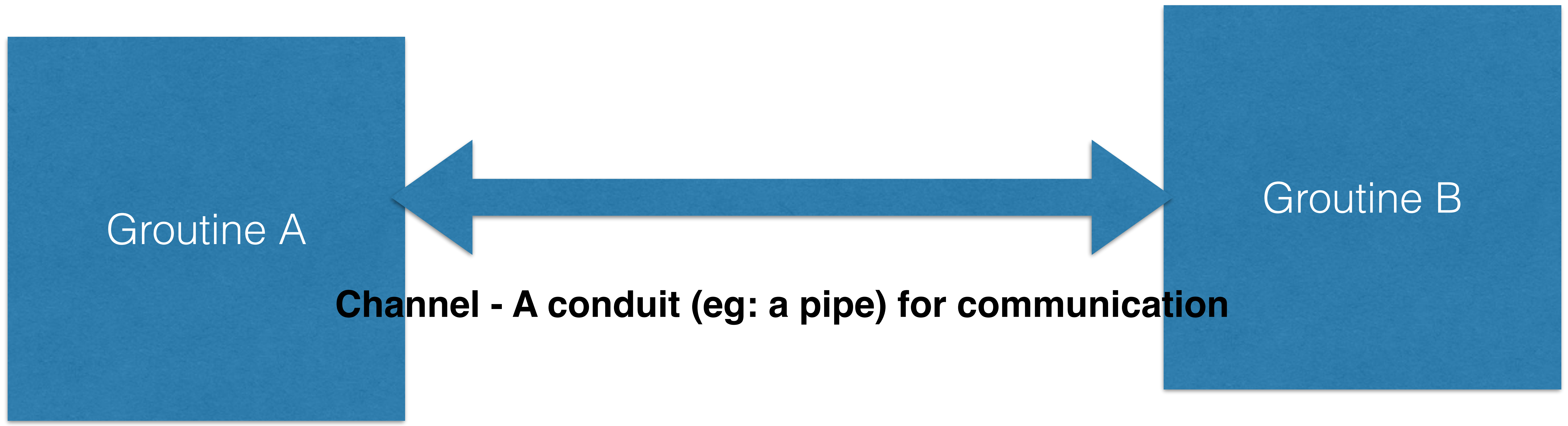
```go
func main() {
   go foo()
   go bar()
    // continue doing other things
}

func foo() {
    fmt.Println("foo")
}

func bar() {
    fmt.Println("bar")
}
```

Channels

Channels provide a simple mechanism for goroutines to communicate, and a powerful construct to build sophisticated concurrency patterns.



Channel - A conduit (eg: a pipe) for communication

Channel Operations:

Send
Receive

Channels

- Channels are goroutine safe
- Channels carry messages between goroutines
- Channels cause blocking and unblocking

Types of Channel

- UnBuffered Channel
- Buffered Channel

Unbuffered Channel

- Synchronous communication
- Block sending goroutine or receiving goroutine until corresponding send or receive happening on the same channel

Buffered Channel

- Asynchronous communication
- Stores up to capacity elements and provides FIFO semantics

- Concurrency in Go is the ability for functions to run independent of each other.
- Go provides concurrency using two paradigms: goroutine and channel.
- A concurrently executing function in Go, is known as goroutine.
- Channel is the communication mechanism to send and receive data between goroutines.
- There are two types of channels are available: unbuffered channel and buffered channel.
- The built-in function make is used to initialize channels.

- Channels have two principal operations: Send, which writes message to channel, and Receive, which receives messages from channels.
- Once a channel is closed using built-in close function, no more values can be sent into the channel.
- The unbuffered channel is used to perform synchronous communication between goroutines and buffered channel is used for perform asynchronous communication.
- The unbuffered channel provides the exchange of data in a synchronous manner that ensures that a send operation on the channel from one goroutine, would be successfully delivered to another goroutine with a corresponding receive operation on the same channel.

- A buffered channel is capable of buffering values up to its capacity and it provides asynchronous communication for data exchange.

- A buffered channel is like a queue on which a send operation doesn't block any goroutine because of its capability for holding elements. The capacity of a buffered channel is determined when it is created by using make function.

- A receive operation on the closed channel receives default value of the channel’s element type (eg: empty string for a channel of string)

- The Select expression is used make channel operations between multiple channels.

HTTP Programming

HTTP Programming Model

- A Request - Response Programming model
  - HTTP Client sends a request to the server
  - Server serves the request and sends back a response
- In Go, standard library package ***net/http***, provides the essential functionalities for writing HTTP applications.

Major components of net/http

- **ServeMux** (struct): HTTP request multiplexer (router)
- **Handler** (interface): Serves the HTTP requests. Writes headers and bodies into HTTP response

Handler interface

```
 type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

type ResponseWriter interface {
 }

type Request struct {
 }
```

HandlerFunc

HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers.

```
type HandlerFunc func(ResponseWriter, *Request)

func (h HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {

}

func index(w ResponseWriter, r *Request) {}

// Create HandlerFunc value

handler := HandlerFunc (index)
```

HTTP Middleware

- Pluggable and self-contained piece of code that wraps web application handlers.
- Components that work as another layer in the request handling cycle, which can execute some logic before or after executing your HTTP application handlers.
- Great for implementing cross-cutting concerns: Authentication, authorization, caching, logging, etc.

```
func middlewareHandler(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    // Our middleware logic goes here before executing application handler
        next.ServeHTTP(w, r)
    // Our middleware logic goes here after executing application handler
})
}
```

Package Context

- Package context defines the Context type, which carries deadlines, cancelation signals, and other request-scoped values across API boundaries and between processes.
- Context created using:
  - WithCancel
  - WithDeadline
  - WithTimeout
  - WithValue

```go
func DoSomething(ctx context.Context, arg Arg) error {

}
```

- Do not pass a nil Context, even if a function permits it. Pass context.TODO if you are unsure about which Context to use.
- Use context Values only for request-scoped data that transits processes and APIs, not for passing optional parameters to functions.
- The same Context may be passed to functions running in different goroutines; Contexts are safe for simultaneous use by multiple goroutines.

Thanks

Shiju Varghese
Senior Consulting Solutions Architect
https://shijuvar.medium.com
https://github.com/shijuvar
https://linkedin.com/in/shijuvar
https://instagram.com/shijuvar

Shiju Varghese's Masterclass: bit.ly/shiju-go