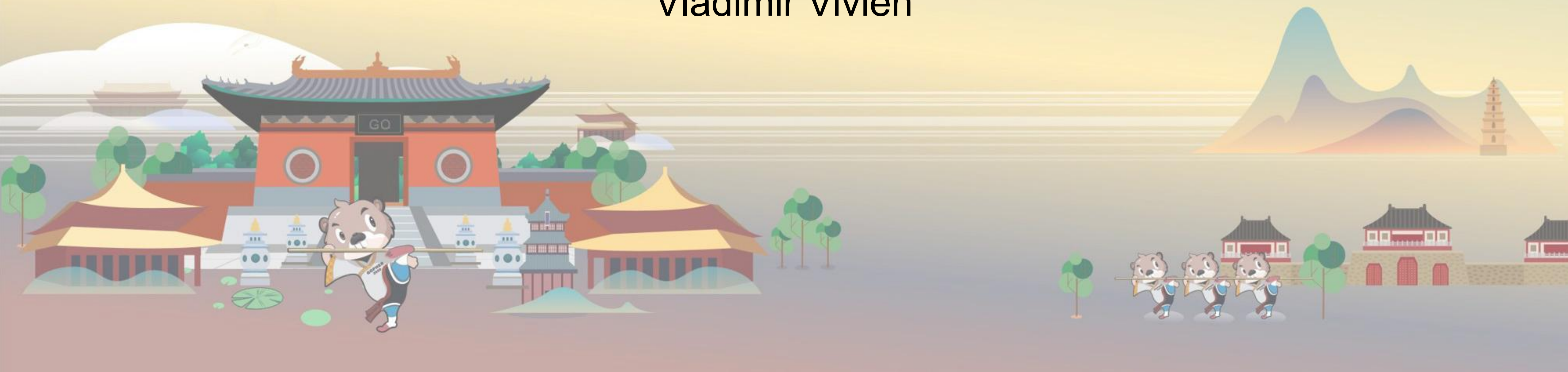
GOPHER CHINA 2020

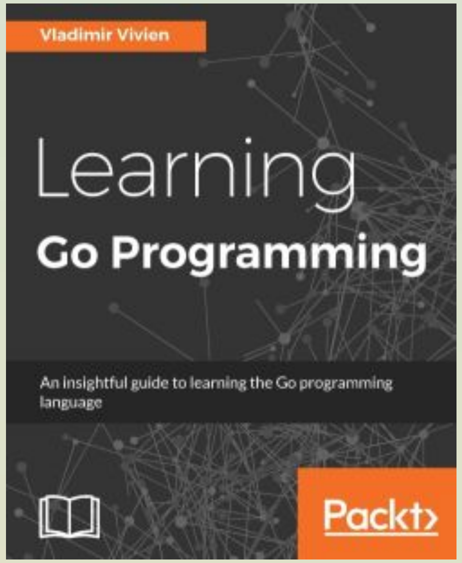
中国 上海 / 2020-11.21-22

Go to the Cloud
Why do people building cloud computing choose Go?

Vladimir Vivien



About Me



Vladimir Vivien (@VladimirVivien)
Staff Engineer, VMware
Go book author, technologist

What is Cloud Native Computing?

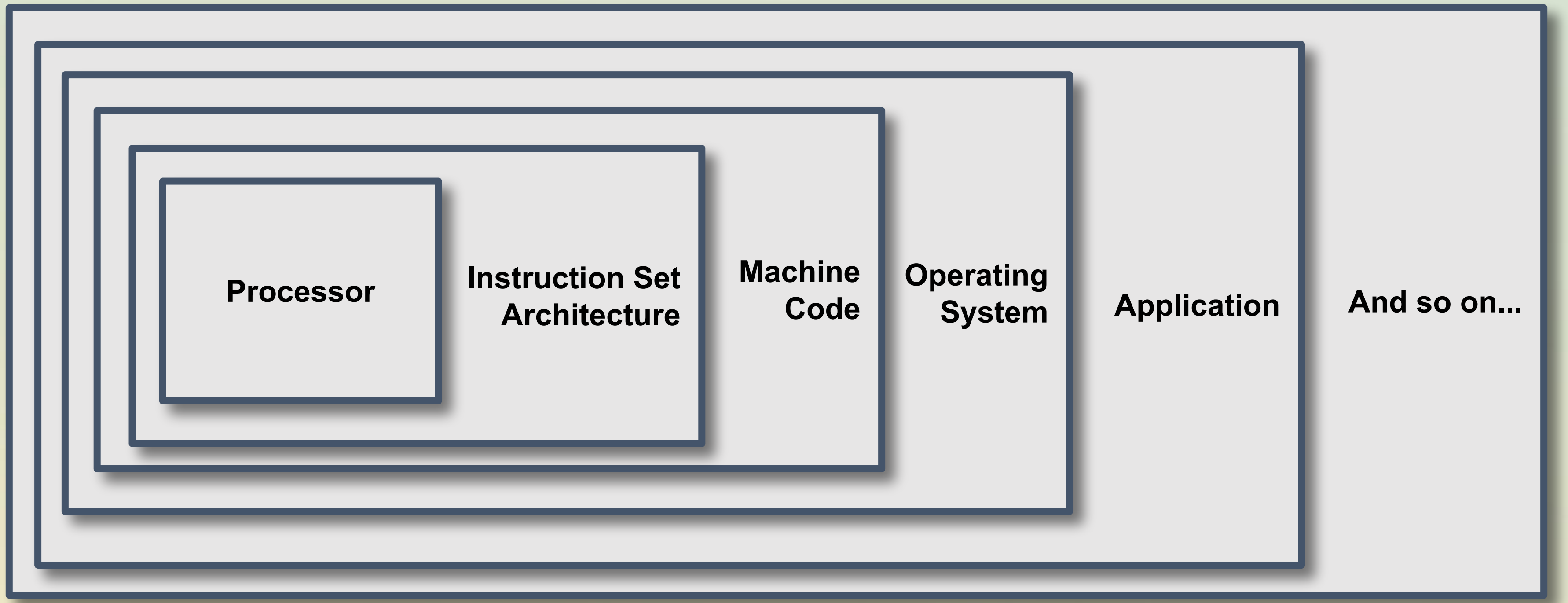
Cloud Native Computing

A collection of technologies that automates the deployment and management of highly decoupled and resilient application workloads across a uniform abstraction of compute resources to create dynamic and elastic infrastructures.

What makes cloud native computing possible?

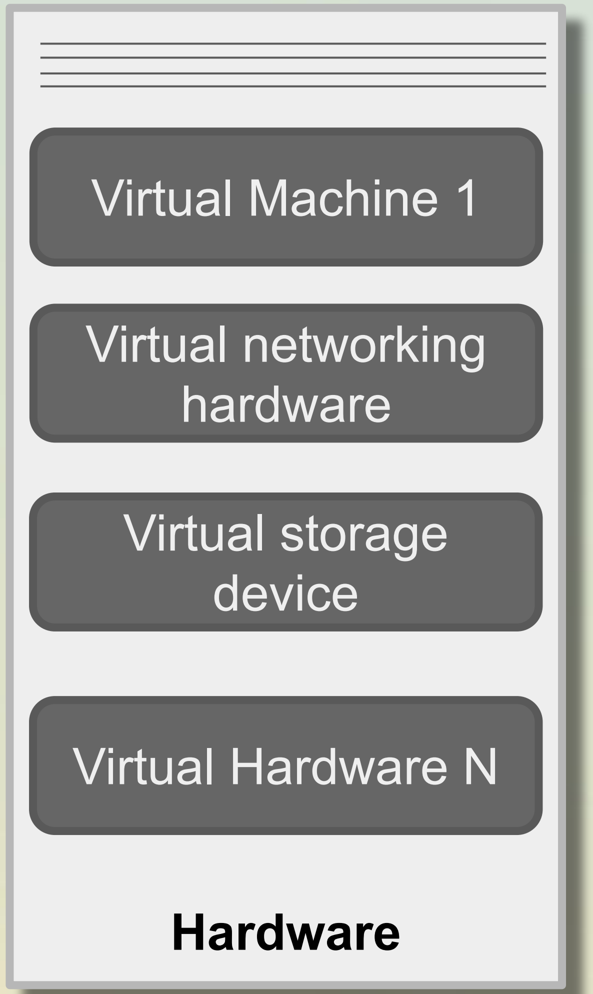
Abstraction

The ability of one technology to make it easier to use another technology by hiding its complexities and/or simplifying its representation.


Processor
Instruction Set Architecture
Machine Code
Operating System
Application
And so on...

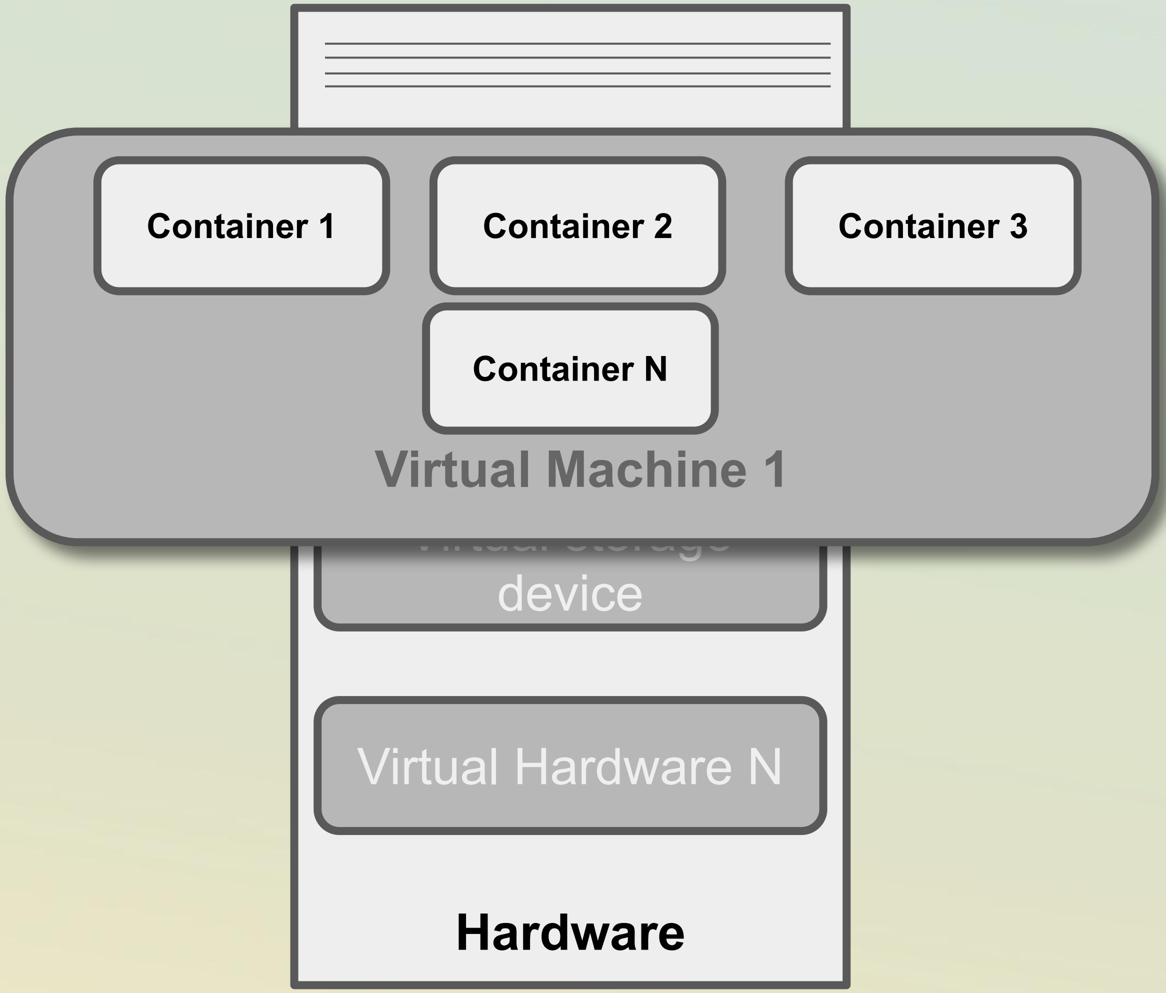
Important abstraction 1:

Hardware as software
(VMs, storage, networking, etc)



Important abstraction 2:

The containerized process



Abstraction leads to automation.

Automation leads to:

Programmability

Repeatability

Resiliency

Cloud Native Computing Attributes

Cloud native infrastructures

- Abstraction of compute, storage, and networking resources
- Virtualized resources can be programmed via well-defined API endpoints that are consumable over HTTP
- Leading to automation of resource creation, observation, management, and resource teardown
- Creates an elastic infrastructures that are capable of dynamically scale across resources

Cloud native applications

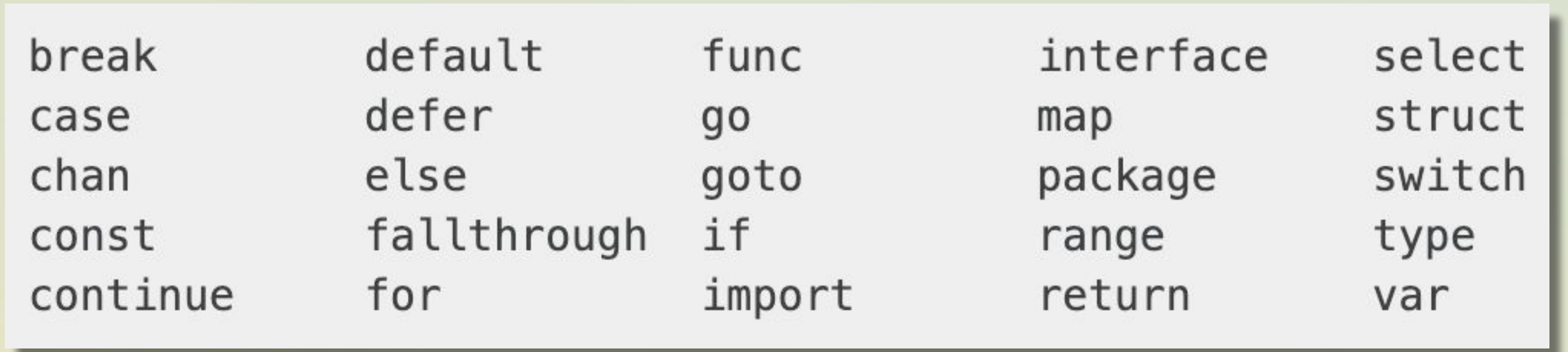
- Built as self-contained, portable, and executable resources
- Single binaries packaged to run in lightweight containers
- Built with automated tools for agile deployments
- Run as loosely-coupled services cross-cutting different domains
- Services use HTTP to expose well-defined API endpoints
- Ability to scale and move across compute resources as needed

Why Choose Go for Cloud Native Computing?

Simplicity

Go has a small language spec and borrows well-known primitives from other languages.

Go keywords (all of them)



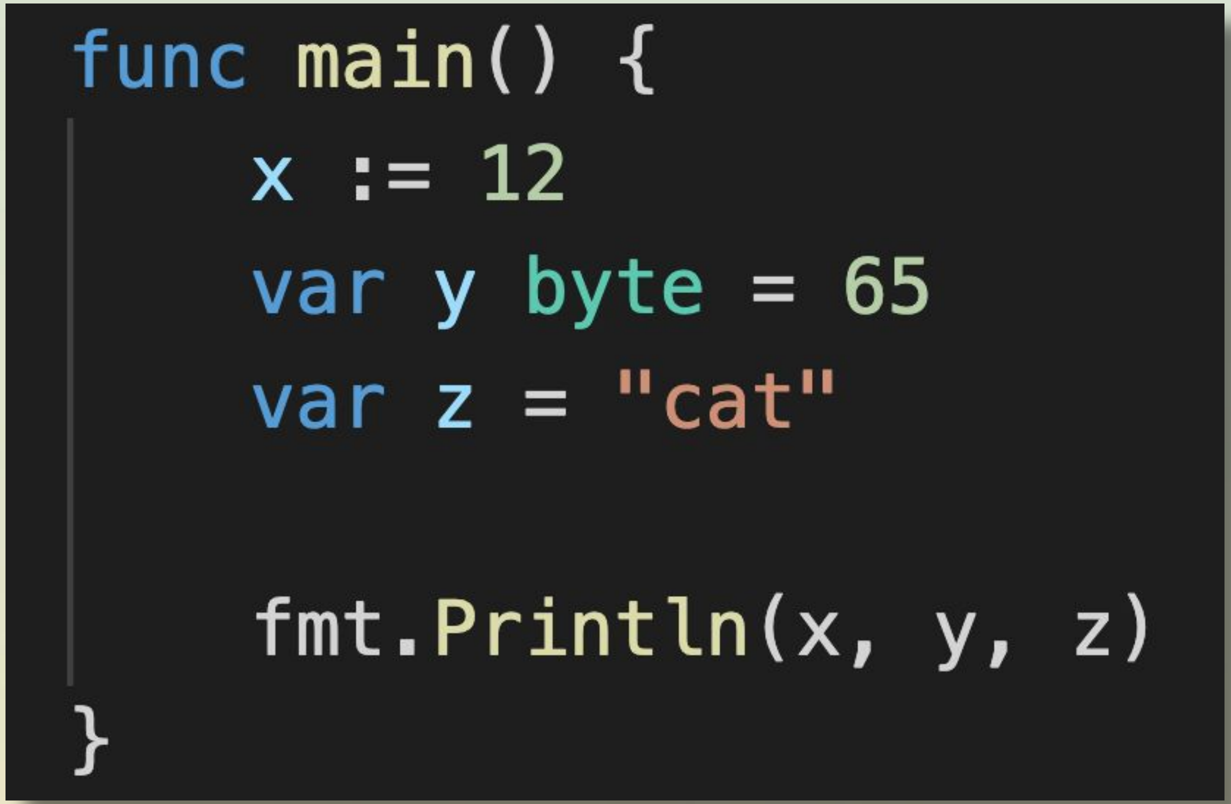
```
break        default      func         interface    select
case         defer        go           map          struct
chan         else         goto         package      switch
const        fallthrough  if           range        type
continue     for          import       return       var
```

Simple syntax

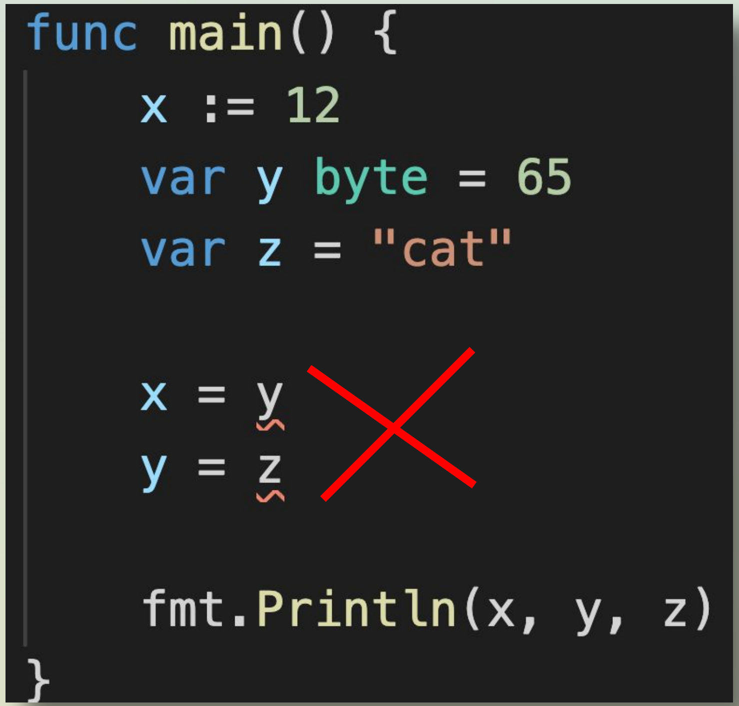
- Go source code is easy to read
- Newcomers from other languages can learn Go and be productive quickly
- Projects can scale and grow with new members
- Though simplistic in syntax, the Go language allow developers to construct large and complex cloud native systems such as Docker, Kubernetes, Prometheus, Consul, etcd, etc.

Strongly typed

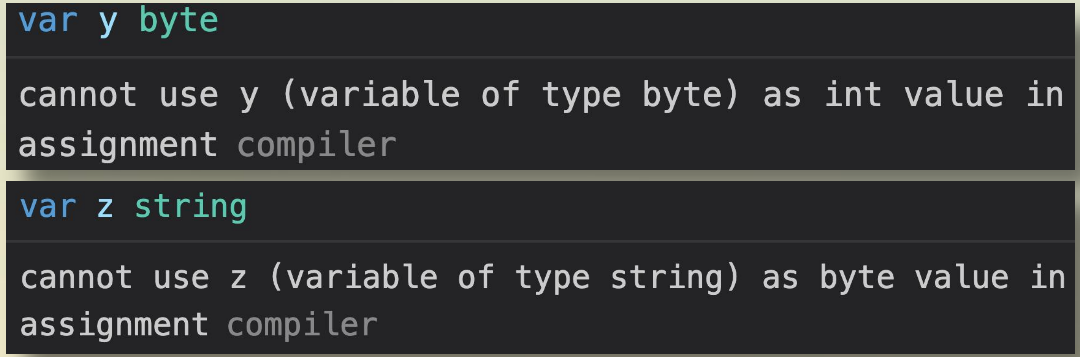
Go has a powerful type system where all values are required to be typed (implicitly/explicitly) with applied compile-time checks and safety.



```go
func main() {
	x := 12
	var y byte = 65
	var z = "cat"

	fmt.Println(x, y, z)
}
```



```go
func main() {
	x := 12
	var y byte = 65
	var z = "cat"

	x = y
	y = z

	fmt.Println(x, y, z)
}
```



```
var y byte
cannot use y (variable of type byte) as int value in
assignment compiler
```

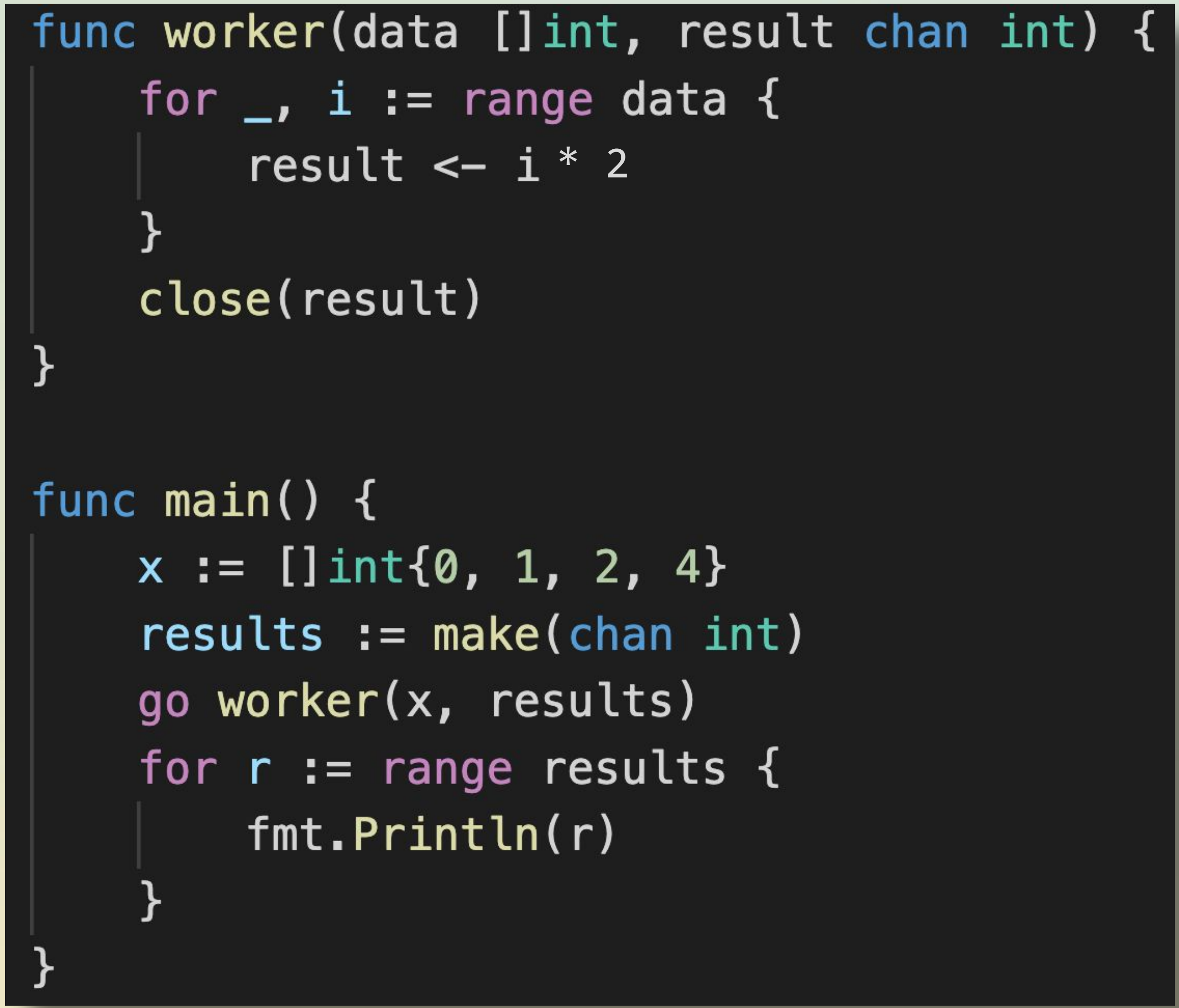
```
var z string
cannot use z (variable of type string) as byte value in
assignment compiler
```

Type safety

- Supports data type stability in programs at runtime
- The type safety of a program can be verified at compiled time with the ability for programs to scale in size and complexity without data type correctness and safety issues
- This eliminates entire classes of runtime errors that are caused by data and type hazards

Concurrency

Go uses simple concurrency primitives known as *goroutines* with support for inter-process communication via *channels*.



```go
func worker(data []int, result chan int) {
    for _, i := range data {
        result <- i * 2
    }
    close(result)
}

func main() {
    x := []int{0, 1, 2, 4}
    results := make(chan int)
    go worker(x, results)
    for r := range results {
        fmt.Println(r)
    }
}
```

Concurrent Programming

- Uses simple concurrency constructs that are easy to reason about
- Goroutines can multiplex hundreds or even thousands of concurrently running processes efficiently across hardware threads
- Type safe inter-process communication that can be used for synchronization and/or data sharing
- Easily create complex, multi-process, and highly performant concurrent cloud native programs to handle large scale infrastructures and cloud native application workloads

Memory management

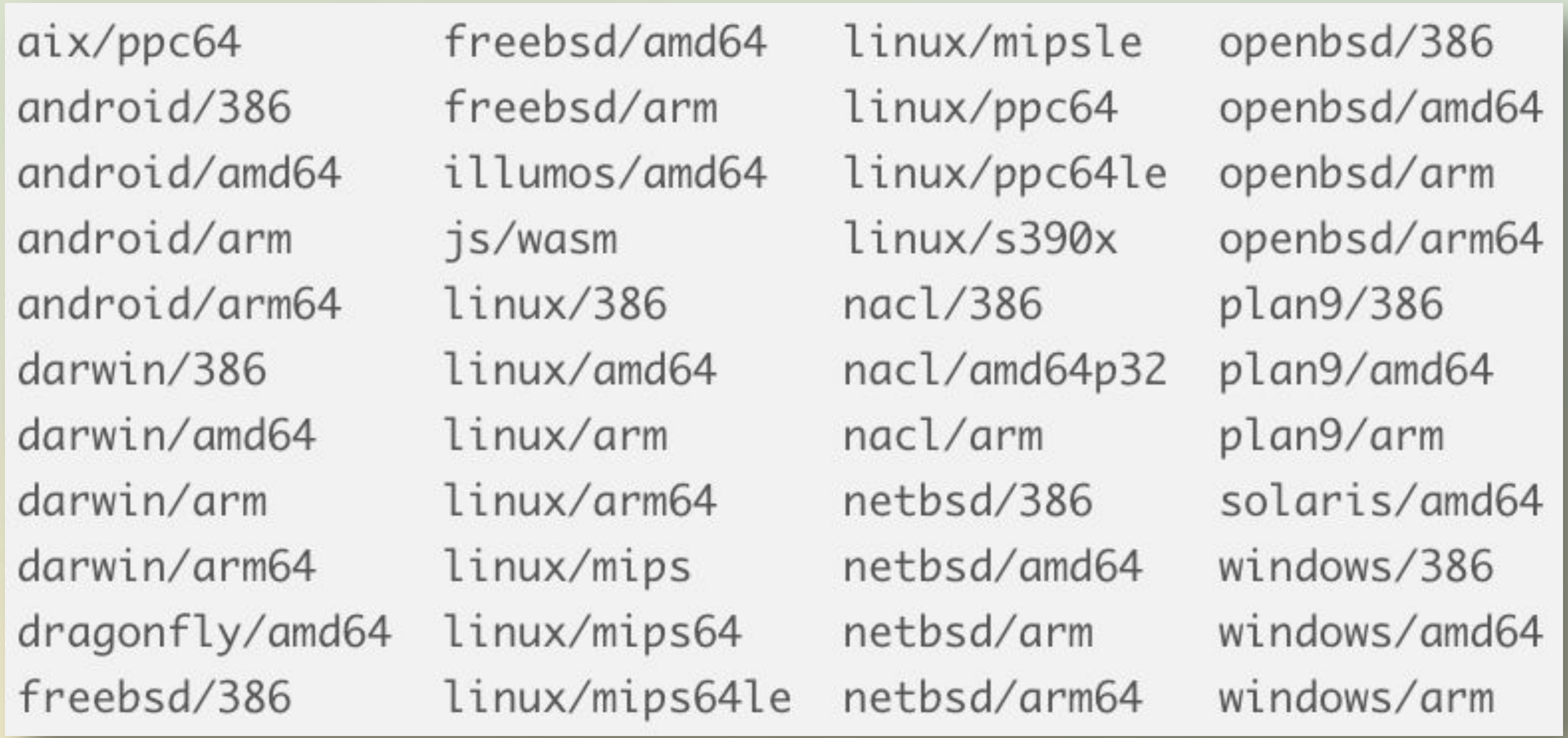
Go programs are compiled to include a runtime manager which, among many things, handles memory allocation and garbage collection.

Automatic memory management

- In Go, programmers are not responsible (and not allowed) to allocate and manipulate memory addresses and values directly
- Memory is allocated and deallocated using the runtime manager.
- Unused memory is automatically collected at runtime once out of scope
- This eliminates an entire class of safety and security related bugs
- Cloud native developers rely on these safeguards to create safe and performant applications and shared infrastructures for running multi-tenant workloads

Cross-platform compilation

The Go compiler can automatically compile a single source code to target multiple OSs and hardware architectures.



```
aix/ppc64        freebsd/amd64   linux/mipsle    openbsd/386
android/386      freebsd/arm     linux/ppc64     openbsd/amd64
android/amd64    illumos/amd64   linux/ppc64le   openbsd/arm
android/arm      js/wasm         linux/s390x     openbsd/arm64
android/arm64    linux/386       nacl/386        plan9/386
darwin/386       linux/amd64     nacl/amd64p32   plan9/amd64
darwin/amd64     linux/arm       nacl/arm        plan9/arm
darwin/arm       linux/arm64     netbsd/386      solaris/amd64
darwin/arm64     linux/mips      netbsd/amd64    windows/386
dragonfly/amd64  linux/mips64    netbsd/arm      windows/amd64
freebsd/386      linux/mips64le  netbsd/arm64    windows/arm
```

https://www.digitalocean.com/community/tutorials/building-go-applications-for-different-operating-systems-and-architectures

```
env GOOS=windows GOARCH=amd64 go build github.com/vladimirvivien/echo
```

Building for multiple platforms

- The Go toolchain can compile programs to automatically target a combination of several operating systems and hardware architectures from a single source code
- This plays well in cloud native computing environments allowing developers to automatically build programs to target virtualized resources, which may be running a variety of operating systems and architectures

Static linking

Go programs are compiled into a single statically-linked binary.

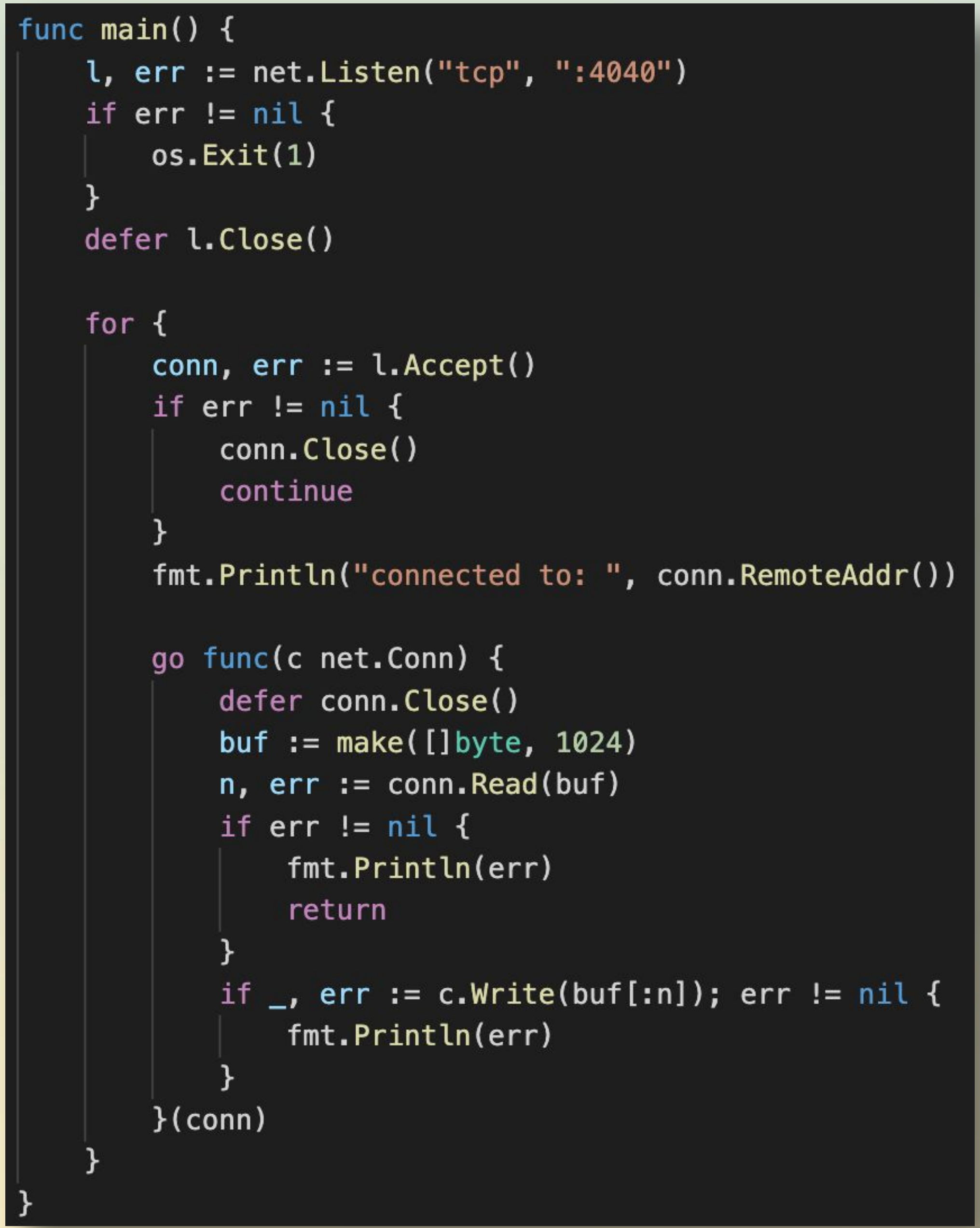
The single binary

- By default, Go programs are compiled into a single statically linked binary with no external dependencies needed to run.
- Once compiled, the program is ready to run, eliminating an entire set of issues that can arise with runtime dependency management.
- In cloud native computing systems, this features allows Go programs to be packaged as lightweight containers that can be quickly be retrieved and ready to run, making them ideal for orchestrated environments such as Kubernetes.

Network programming

The Go standard library comes with inherent support for low-level network programming and multiple protocols.

Simple echo server



```go
func main() {
    l, err := net.Listen("tcp", ":4040")
    if err != nil {
        os.Exit(1)
    }
    defer l.Close()

    for {
        conn, err := l.Accept()
        if err != nil {
            conn.Close()
            continue
        }
        fmt.Println("connected to: ", conn.RemoteAddr())

        go func(c net.Conn) {
            defer conn.Close()
            buf := make([]byte, 1024)
            n, err := conn.Read(buf)
            if err != nil {
                fmt.Println(err)
                return
            }
            if _, err := c.Write(buf[:n]); err != nil {
                fmt.Println(err)
            }
        }(conn)
    }
}
```

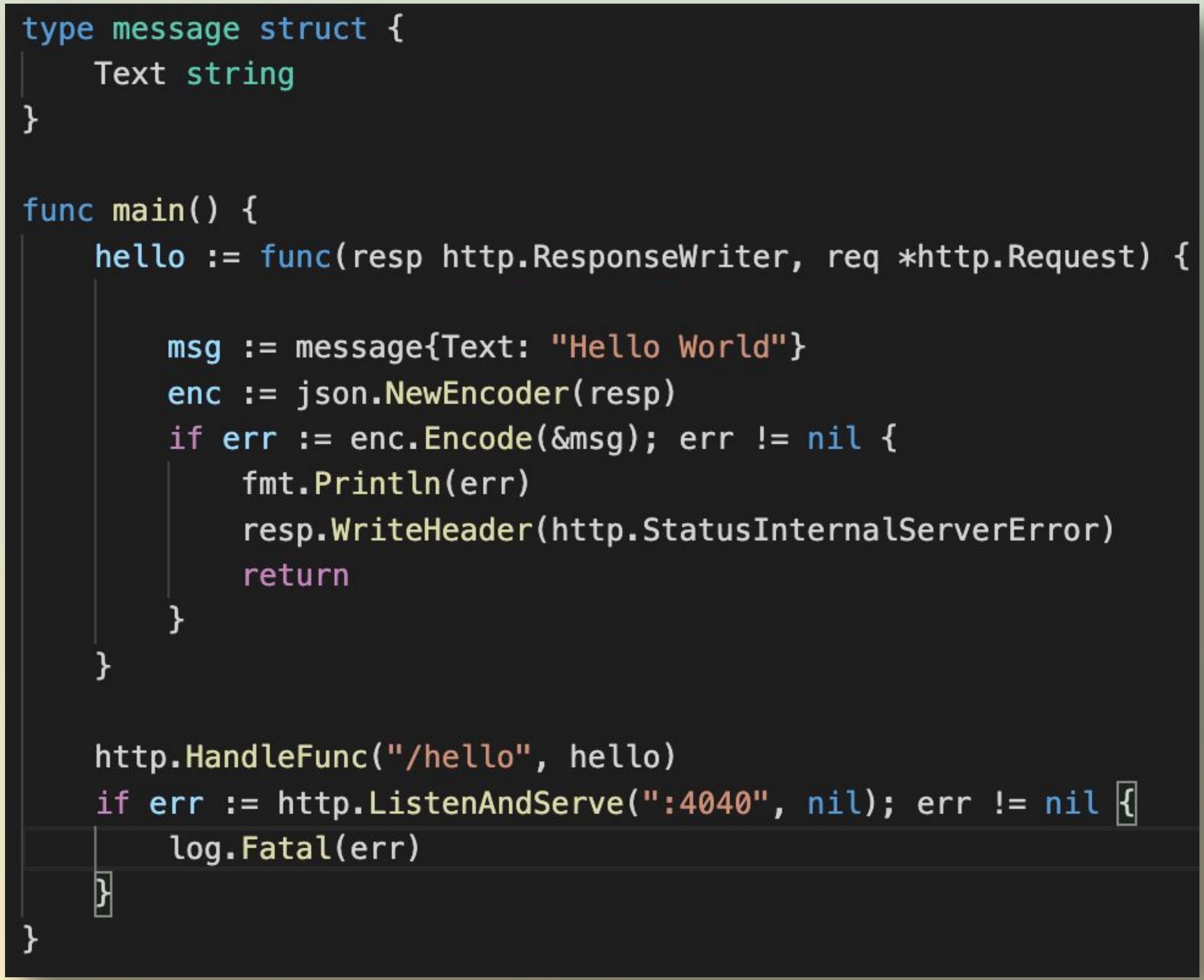
Low-level Networking

- Cloud native infrastructures can span across several hundred (or even thousands) of distributed machines and other components
- These components rely on networking protocols such as TCP/IP for communications and services.
- Using the Go standard library programmers can quickly write correct and performant networking code using protocols such as TCP, IP, UDP, Unix Domains sockets with support for both IPv4 and IPv6.

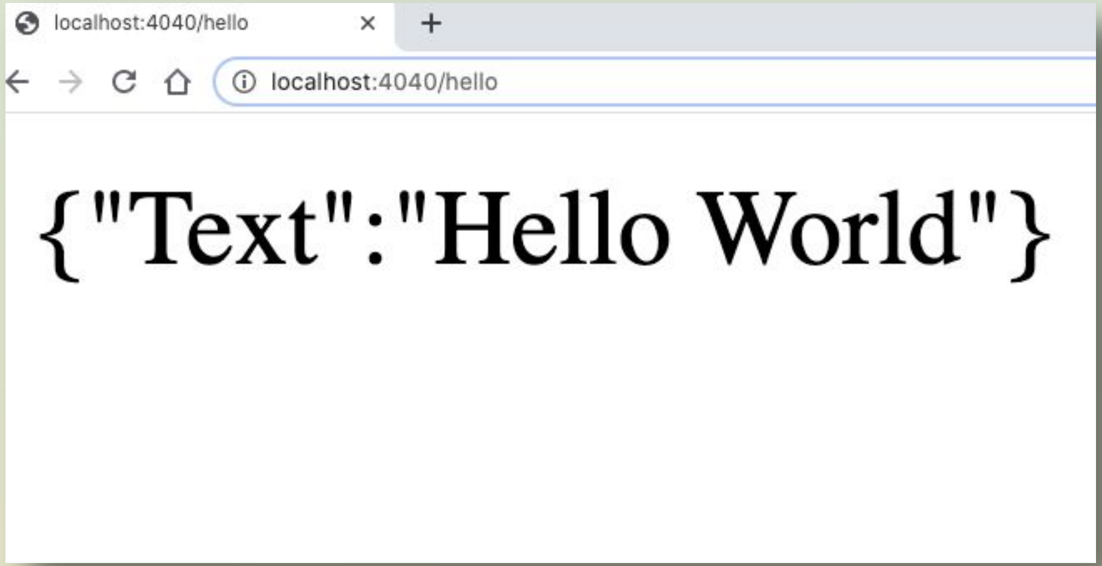
HTTP services

The Go standard library comes with packages to build HTTP-based client and server programs.

Simple HTTP service



```go
type message struct {
    Text string
}

func main() {
    hello := func(resp http.ResponseWriter, req *http.Request) {

        msg := message{Text: "Hello World"}
        enc := json.NewEncoder(resp)
        if err := enc.Encode(&msg); err != nil {
            fmt.Println(err)
            resp.WriteHeader(http.StatusInternalServerError)
            return
        }
    }

    http.HandleFunc("/hello", hello)
    if err := http.ListenAndServe(":4040", nil); err != nil {
        log.Fatal(err)
    }
}
```



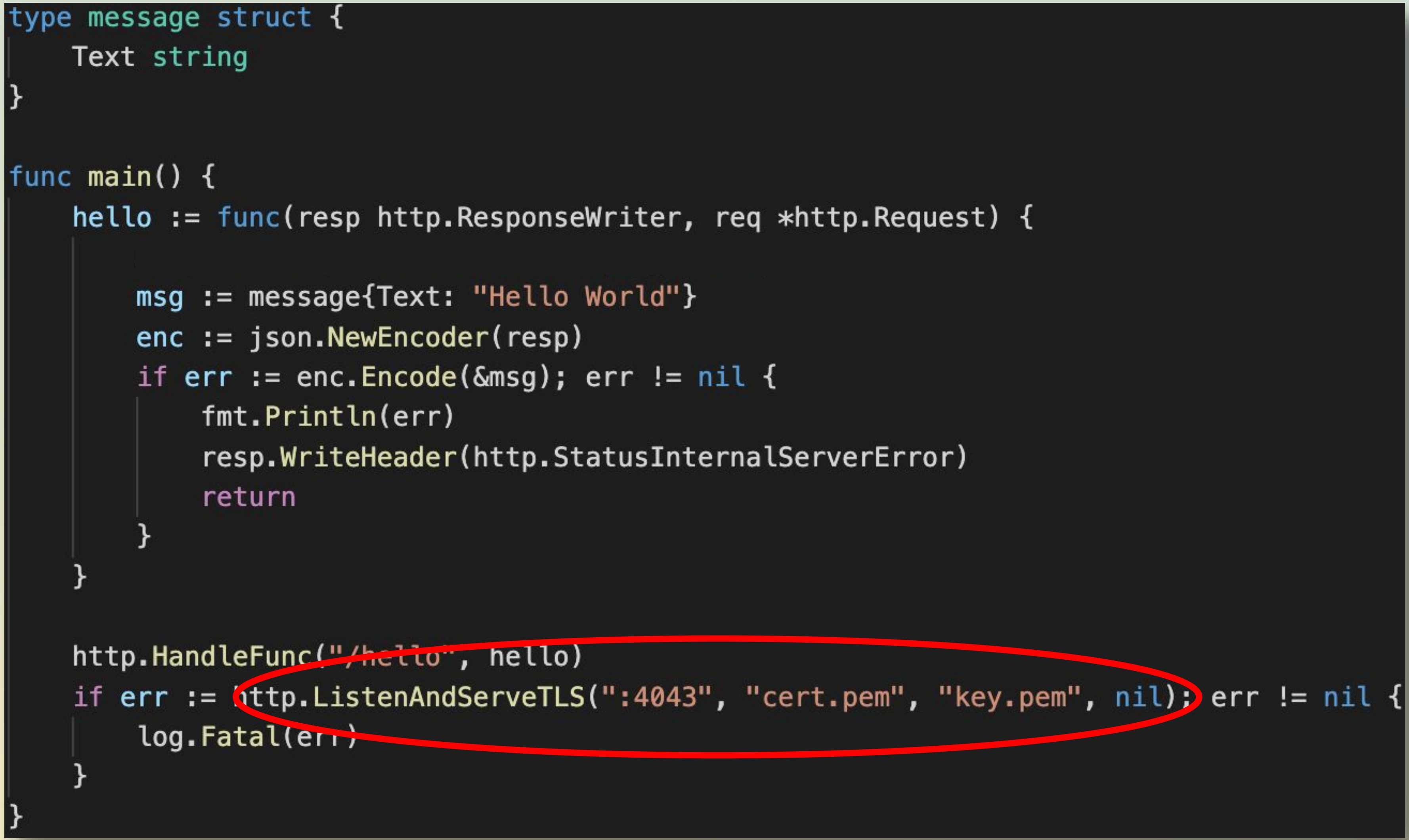
```
{"Text":"Hello World"}
```

Building HTTP service APIs

- Cloud native infrastructures and applications rely heavily on HTTP-based service APIs to facilitate access, operation, observability, and management of computing components and the applications running in these environments
- Fortunately, the Go standard library allows programmers to easily create correct, efficient, performant HTTP server programs to expose robust API endpoints using a variety of protocol including REST and gRPC

Cryptography

The Go standard library provides access to robust implementations of popular cryptographic protocols for building secure programs.



```go
type message struct {
    Text string
}

func main() {
    hello := func(resp http.ResponseWriter, req *http.Request) {

        msg := message{Text: "Hello World"}
        enc := json.NewEncoder(resp)
        if err := enc.Encode(&msg); err != nil {
            fmt.Println(err)
            resp.WriteHeader(http.StatusInternalServerError)
            return
        }
    }

    http.HandleFunc("/hello", hello)
    if err := http.ListenAndServeTLS(":4043", "cert.pem", "key.pem", nil); err != nil {
        log.Fatal(err)
    }
}
```

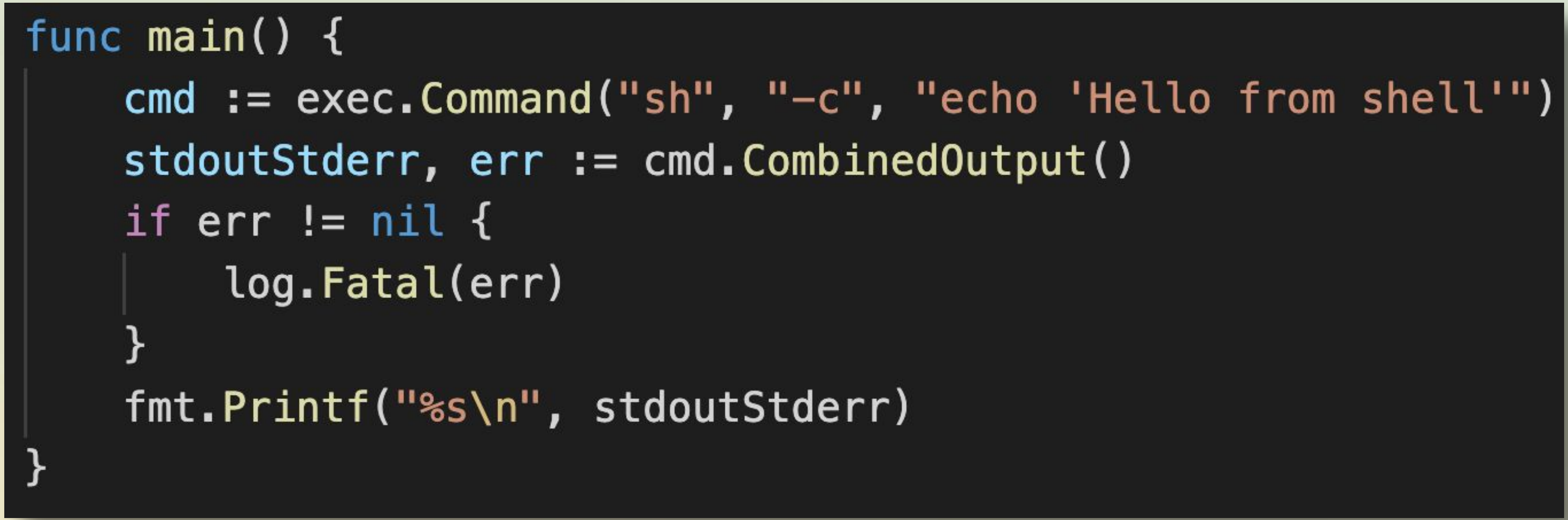
Building secure programs

- Due to its distributed nature, one requirement of cloud native infrastructures is the secure exchange of data between its components to ensure safety and avoid unauthorized data access.
- Using the Go standard library crypto packages programmers can write secure code using a large number of supported cryptographic operations including hash functions, signatures, symmetric/asymmetric key functions, encoder/decoder crypto formats.

Operating system interfacing

Go’s standard library provides a rich set of APIs to create programs that can interface with the enclosing operating system where the programs are running.

Executing a shell command



```go
func main() {
	cmd := exec.Command("sh", "-c", "echo 'Hello from shell'")
	stdoutStderr, err := cmd.CombinedOutput()
	if err != nil {
		log.Fatal(err)
	}
	fmt.Printf("%s\n", stdoutStderr)
}
```

Platform-independent OS interactions

- Cloud native computing requires that infrastructure resources are programmable in order to achieve high level of automation using and accessing functionalities of the underlying operating of compute components for control and management.
- The Go standard library provides packages that make it easy to write platform-independent code that can efficiently interact with the underlying operating system and access functionalities such as file operations, system signaling, and process management.

Go is not a perfect language, but it is perfect for cloud native computing.

Than you

@vladimirvivien