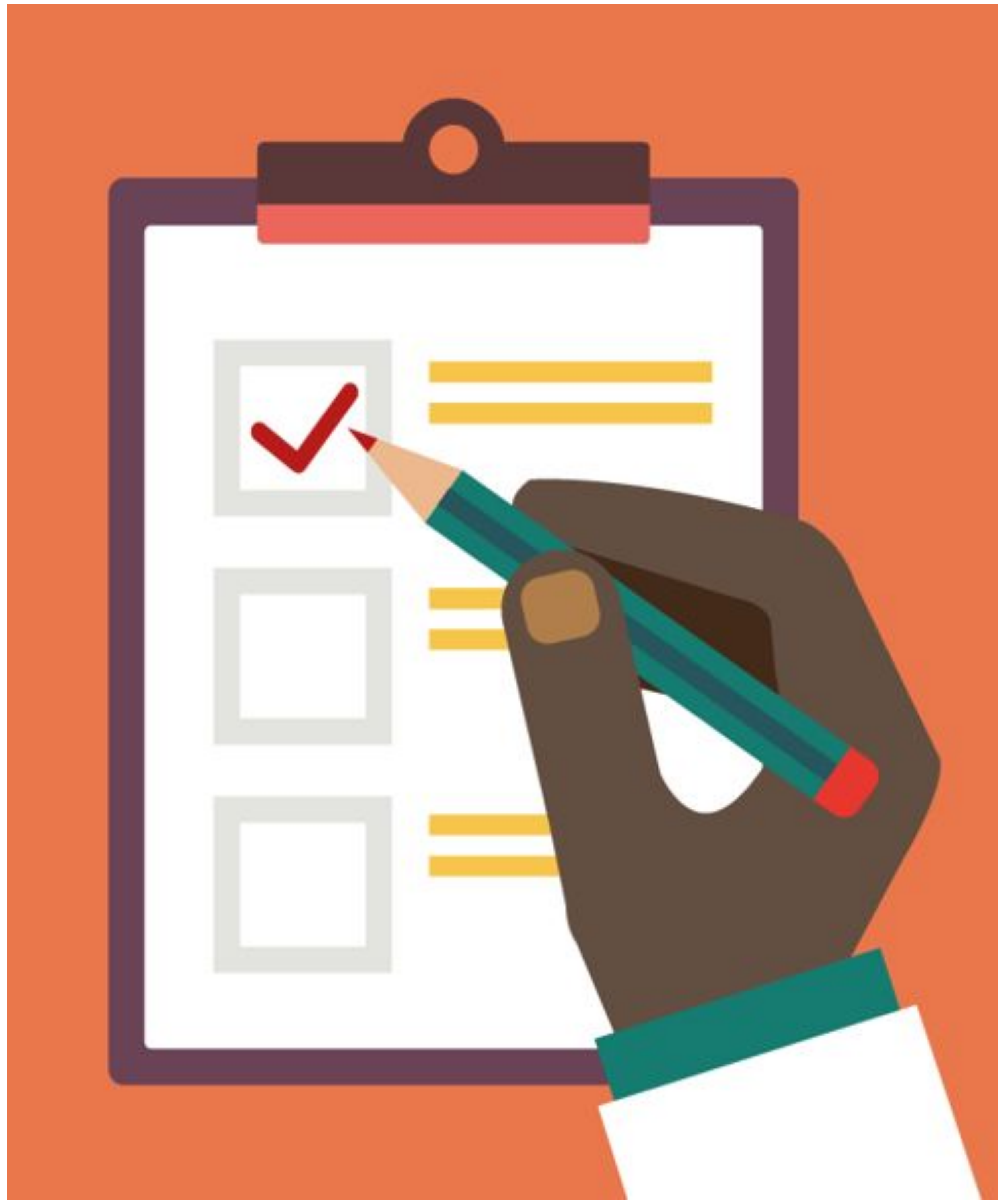
Welcome To

JOSH

DISRUPT YOUR INDUSTRY

Topics to be covered…



- Error Handling
- Defer
- Panic & Recovery
- Context
- Go modules

Error Handling

- Go handles failures by returning error values, not raising exceptions
- Errors in Golang are values, just like integers or string.

```
func main() {
  _, err := Divide(20, 0)
  if err != nil {
    fmt.Println("Error Occured :", err)
  }
}


func Divide(a, b int) (int, error) {
  if b == 0 {
    return 0, fmt.Errorf("can't divide '%d' by zero", a)
  }
  return a / b, nil
}
```

Error Handling

- **error** type is used as standard error in Go

```
type error interface {
    Error() string
}
```

- **error** is an interface type, so zero value is nil

Constructing Errors

- error type object can be created using built-in packages **errors** and **fmt**

```
errors.New("error message")
```

```
fmt.Errorf("error message with formatting, %s", val)
```

Run in Go Playground

Error Handling

Defining Expected Errors

Giving identity to an error and using it in specific cases and at specific places

```
var ErrDivideByZero = errors.New("divide by zero")
```

Run in Go Playground

Compare Two Errors

```
errors.Is(err, target error) bool
```

Error Handling

Defining Custom Error Types

- error type is a built-in interface with a single method Error() string.
- All rules governing interfaces apply to error type.
- We can make our custom type implement the error interface

Run in Go Playground

Assign target with error value

`errors.As(err error, target any) bool`

Error Handling

Good Practices

- Always check returned errors.
- Early return at place of if-else with error values.
- If a function returns an error, it should always be the last returned value.
- Define and use expected errors.
- Return zero values of other params, if error is not nil
- Write good error message. It helps with debugging

Defer

- A defer statement postpones the execution of a function until the surrounding function returns.

syntax : `defer function_call()`

```
func main() {
    defer fmt.Println("World")
    fmt.Println("Hello")
}
```

Output :

```
Hello
World
```

[Run in Go Playground]

Defer

Order of Execution :

- A defer statement adds the function call following the **defer** keyword onto a stack.

  i.e **Last-In-First-Out** execution sequence will be followed if multiple defer calls

Run in Go Playground

```
func main() {
    fmt.Println("Hello")
    for i := 1; i <= 3; i++ {
        defer fmt.Println(i)
    }
    fmt.Println("World")
}
```

Output :

```
Hello
World
3
2
1
```

Defer

Argument evaluation & Order of Execution :

- The deferred call's arguments are evaluated immediately, even though the function call is not executed until the surrounding function returns.

  Run in Go Playground

When are defer statement executed?

- Surrounding function finishes execution

- When the enclosing function - return statement is encountered
- When the enclosing function - falls off
- When the enclosing function - panics

  Run in Go Playground

Defer

- Deferred anonymous functions may access and modify the surrounding function’s named return parameters.

```
func foo() (result string) {
    defer func() {
        result = "Change World" // change value at the very last moment
    }()
    return "Hello World"

}
```

`Value of result variable :` `“Change World”`

Run in Go Playground

Defer

Anonymous functions with defer :

- Using Outside Variables inside defer function

```
func main() {
    for i := 0; i < 2; i++ {
        defer func() {
            fmt.Printf("%d\n", i)
        }()
    }
}
```

Output :

```
2
2
```

Run in Go Playground

Defer

Use Cases :

- Defer is often used to perform clean-up actions, such as closing a file or unlocking a mutex.
- Ideally, defer statement should follow the Open action. This helps us achieve desired code clearity.
- Handle panic recovery

```
func main() {
f := createFile("/tmp/defer.txt")
    defer closeFile(f)
    writeFile(f)
}
```

Panic & Recover

In Go **panic** and **recover** are technically similar to exception handling in languages like Java or Python.

- panic is equivalent of throw or raise
- recover fills the role of catch

It is always encouraged to use error handling to work with abnormal situations and errors. Usage of panics is generally discouraged.

Panic

What happens when a panic is encountered?

- Program stops execution of function where panic occurred
- All defer functions are executed
- Function returns to caller and to caller function behaves like a call to panic
- process continues up the stack until all functions in the current goroutine have returned
- the panic reaches `main()` and terminates the program
- Stack trace will be printed to stderr

Run in Go Playground

Panic

When does a panic occur?

- Panics can occur at runtime either because of some abnormal condition that we aren’t prepared to handle gracefully or use of built-in **panic(interface{})** function.

- **Common Causes of Panic**

- nil pointer dereference
- Index out of range
- Divide By Zero
- Assignment to nil map and some more

Recover

- Recover attempts to recover from a panic.

- The recover *must* be attempted in a deferred statement as normal execution flow has been halted.

```
func foo() {
   fmt.Println("Start of Foo")

   defer func() {
      if r := recover(); r != nil {
         fmt.Printf("panic occured : %v\n", r)
      }
   }()
   panic("I am panicking")
   fmt.Println("End of Foo")

}
```

Run in Go Playground

Recover

- The **recover()** call will return the argument provided to the initial panic, if the program is currently panicking.
- If the program is not currently panicking, **recover()** will return nil.
- Do not send **nil** as param to panic() function call, otherwise recover() function will take up the nil value and will silently exit.
- If we recover from a panic, we lose the stack trace about the panic. Use **debug.PrintStack()** to get stack trace after recovering.

Run in Go Playground

Panic & Recover

Why Panic and Recover?

- Sometimes it’s easier to propagate an error by panicking and recovering rather than returning errors.

- To get developer attention. error can get ignored sometimes, but panic will always catch the attention

- An unrecoverable error where the program cannot simply continue its execution.

- Recover is needed to stop program from crashing.

Context

- Package `context` in standard library provides type `Context`
- It is when we want to pass around context in our application.

[context]()

Mainly Handles 2 tasks:

- cancellation and propagation
- request scoped values

Context

- We can Create context using 2 functions from package

  - context.TODO()
  - context.Background()

  4 main functions associated with context:

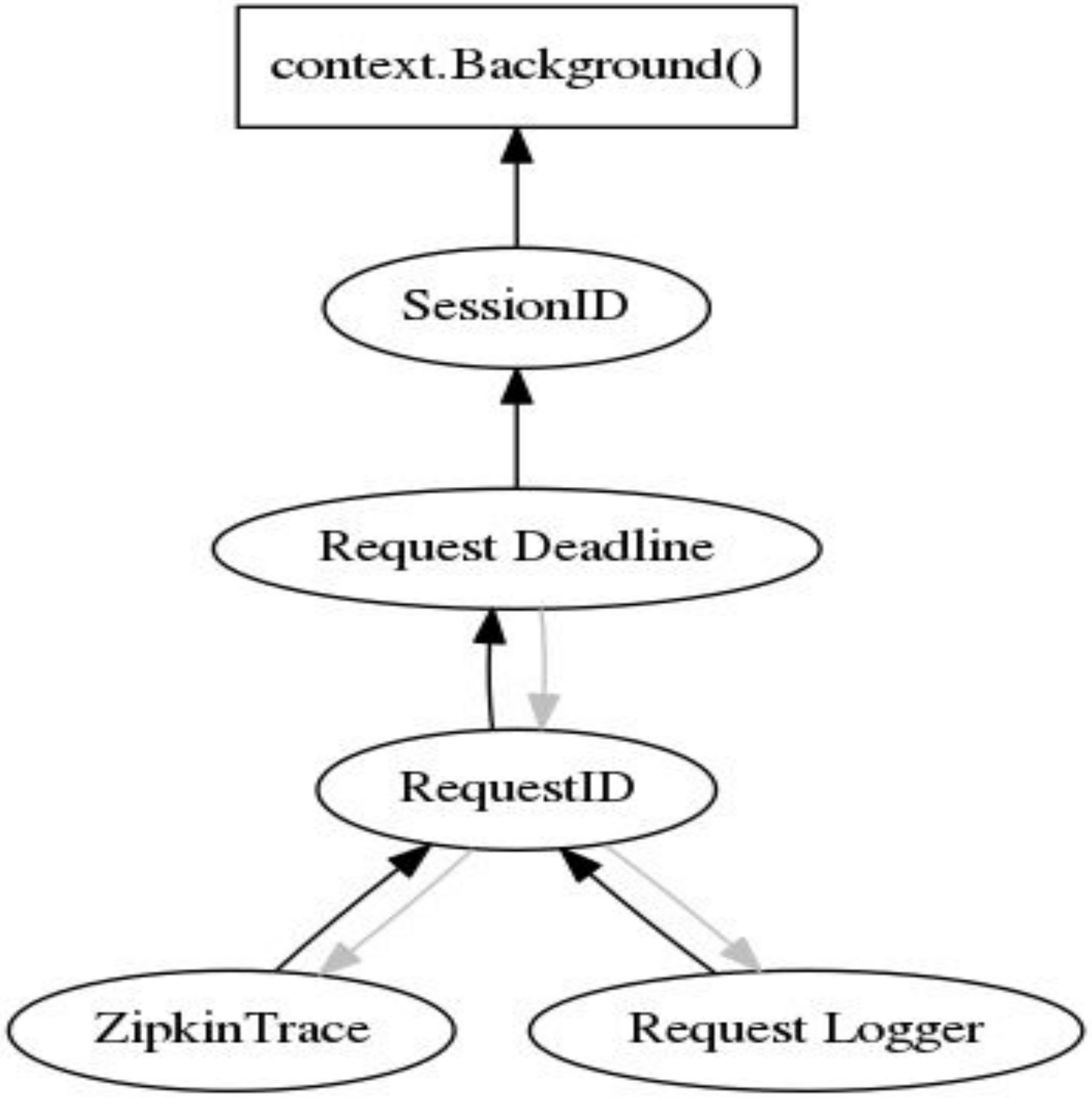
  - WithValue(parent Context, key, val any) Context
  - WithCancel(parent Context) (ctx Context, cancel CancelFunc)
  - WithDeadline(parent Context, d time.Time) (Context, CancelFunc)
  - WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)

Context

- Context is immutable.
- To create a new context, you wrap existing context and add more data.
- You can wrap context multiple time so you can think of it as a tree of values.

Context

```
ctx := context.WithValue(
  context.WithDeadline(
    context.WithValue(
      context.Background(), sidKey, sid),
             time.Now().Add(30 * time.Minute),
         ),
         ridKey, rid,
     )
     trCtx := trace.NewContext(ctx, tr)
 logCtx := myRequestLogging.NewContext(ctx, myRec
```



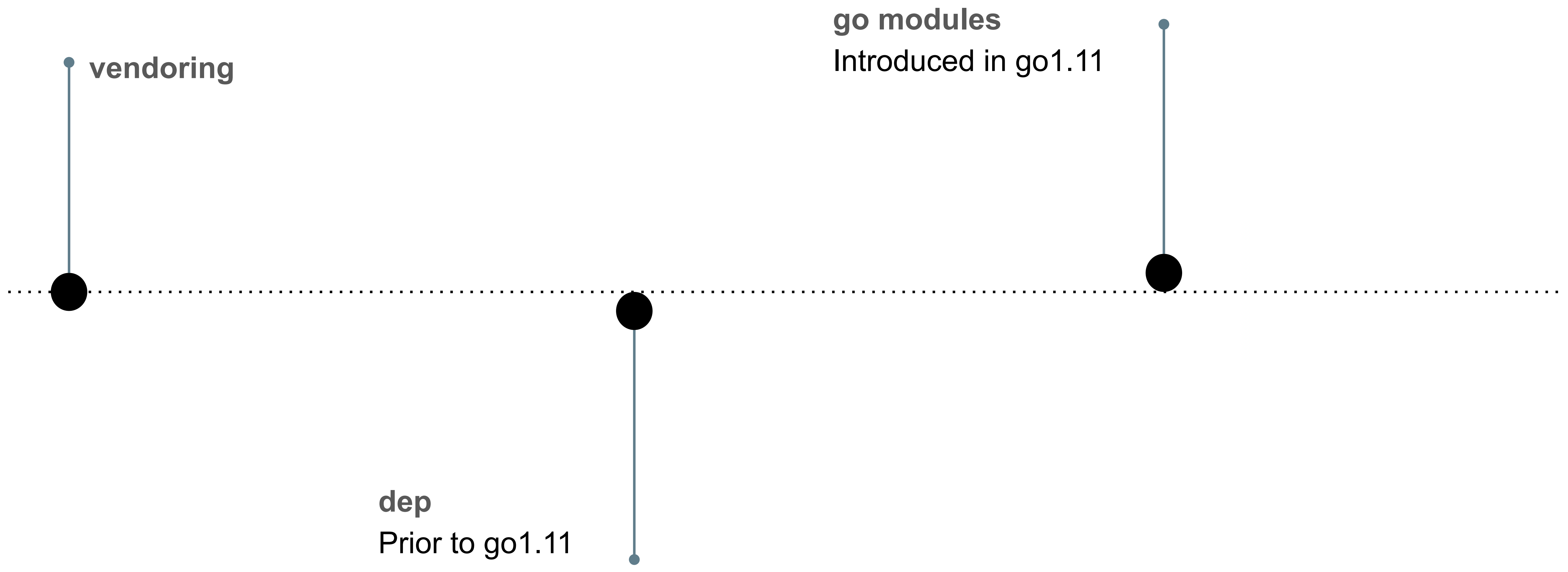
Context

- Context represented as a directed graph.
- Each child context has access to values of its parent contexts.
- Data access flows upwards in the tree (represented by black edges).
- Cancelation signals travel down the tree. If a context is canceled, all of its children are also canceled.
- The cancelation signal flow is represented by the grey edges.

Dependency Management

- What is dependency management?
- Is dependency management essential?
- Does go provides tool for dependency management?

Go Dependency Management



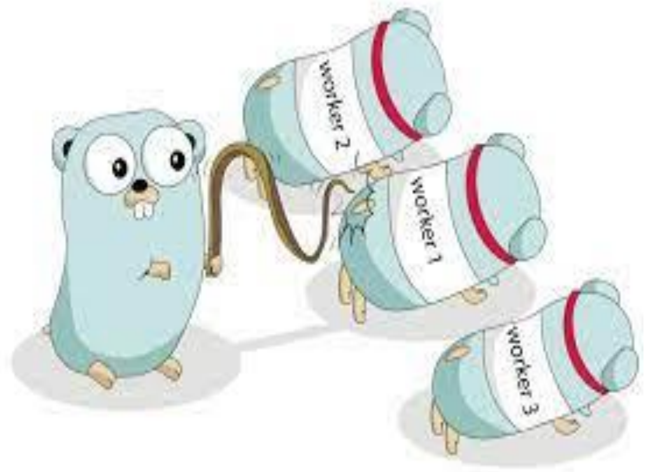
Go Modules

- Go Module is a new dependency management system inbuilt in Go that makes dependency version information explicit and easier to manage.
- Go 1.11 and 1.12 include preliminary support for modules.
- Starting in Go 1.13, module mode is the default for all development.

Important Features of Go Modules

Official Dependency Management Tool

Move Out of $GOPATH



In One single tool

Getting Started with Go modules

1. Create a module -- Write a small module with functions you can call from another module.

2. Call your code from another module -- Import and use your new module.

3. Return and handle an error -- Add simple error handling.

Commands

- **go mod init** creates a new module, initializing the go.mod file that describes it.
- **go build, go test**, and other package-building commands add new dependencies to go.mod as needed.
- **go list -m all** prints the current module's dependencies.
- **go get** changes the required version of a dependency (or adds a new dependency).
- **go mod tidy** removes unused dependencies.

Let’s get some hands-on

Summary of Go modules

approaches	pros	cons
Go modules	• provides go.mod file manage dependencies, very descriptive to understand dependency hierarchy • since it is embedded in language, it comes very handy to implement • upgrading/downgrading of dependencies is very easy • provides feature for warming caches which helps to reduce deployment time • Simple steps to migrate to go modules	Not found any

THANK YOU