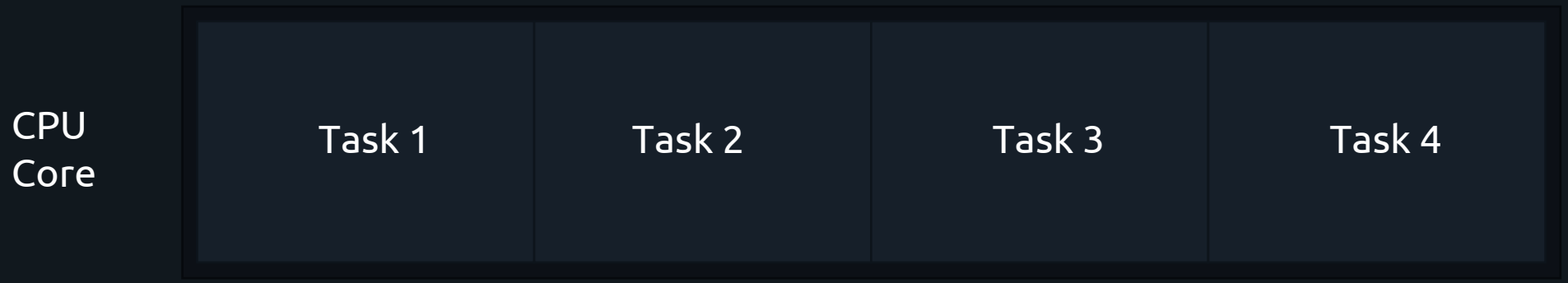
KodeKloud

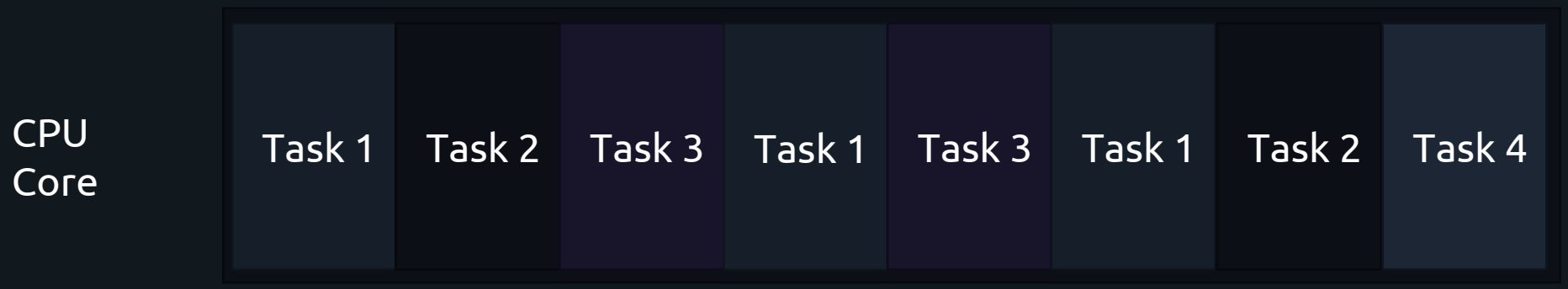
Sequential vs Concurrency

Sequential Processing



Sequential Processing

Concurrent Processing

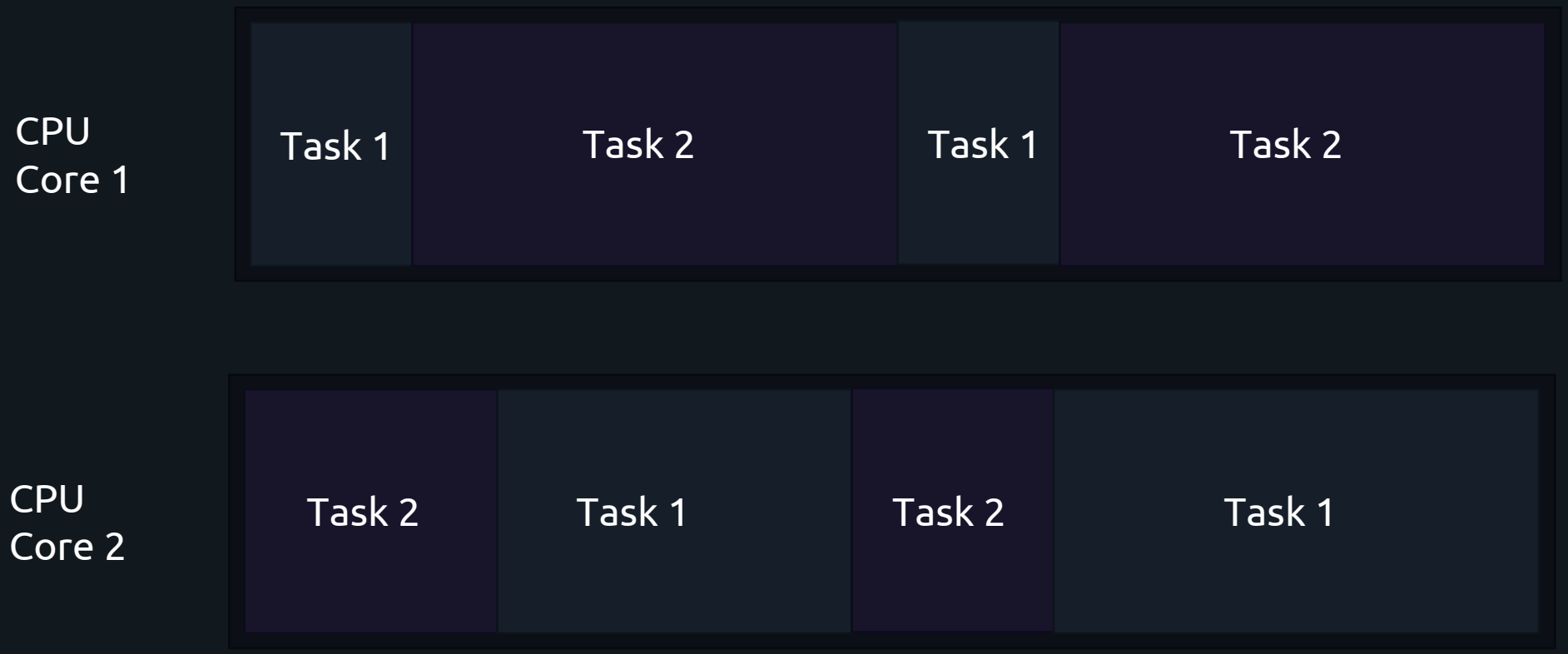


Concurrent Processing

Concurrency is the notion of multiple things happening at the same time.

It is the potential for multiple processes to be in progress at the same time.

Concurrent Processing



Parallel Processing

Examples

Concurrent Processing

user-interactive program

Parallel Processing

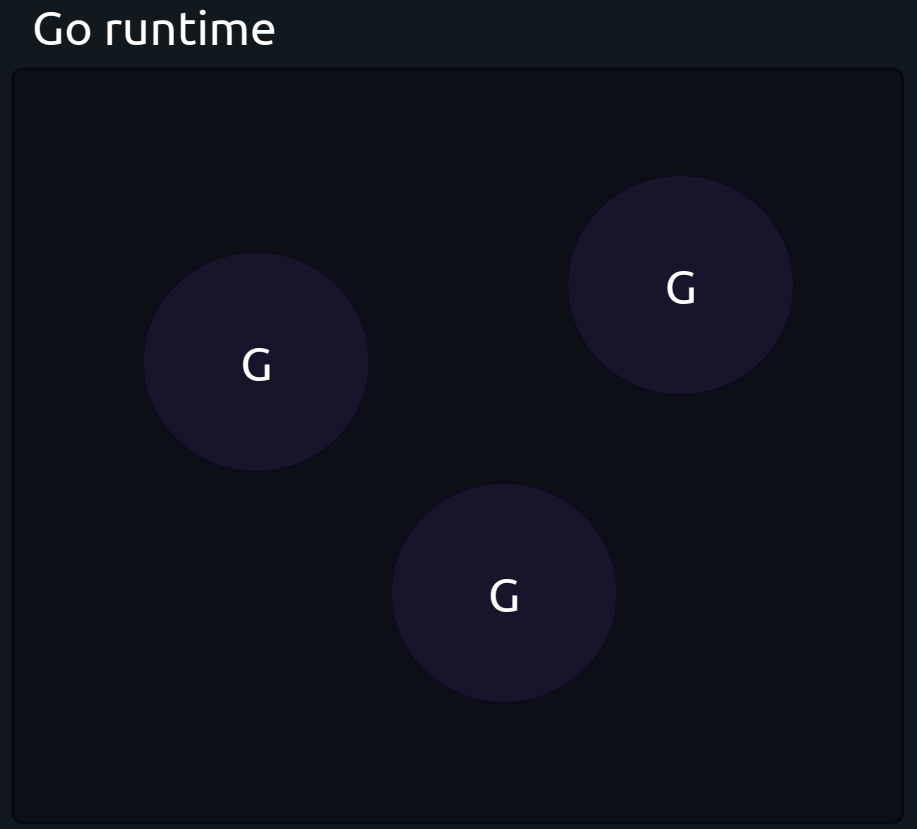
distributed data processing

KodeKloud

Go-routines

Go-routines

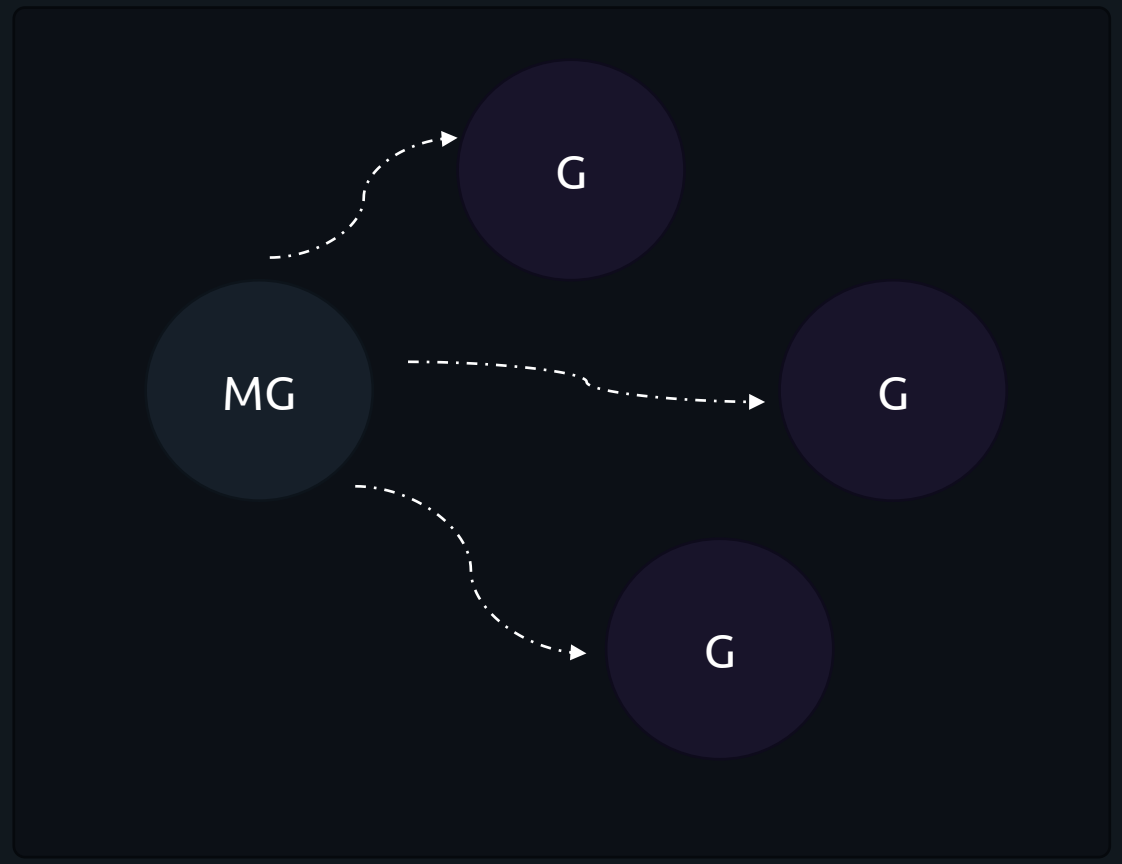
- Considered as a lightweight thread that has a separate independent execution
- Can execute concurrently with other go-routines
- Managed entirely by Go runtime



Syntax

```
go    calculate()
```

Main go-routine



KodeKloud

Anonymous go-routine

- In Golang, anonymous functions are those functions that don't have any name. Simply put, anonymous functions don't use any variables as a name when they are declared.

- Anonymous functions in golang can also be called using go-routine.

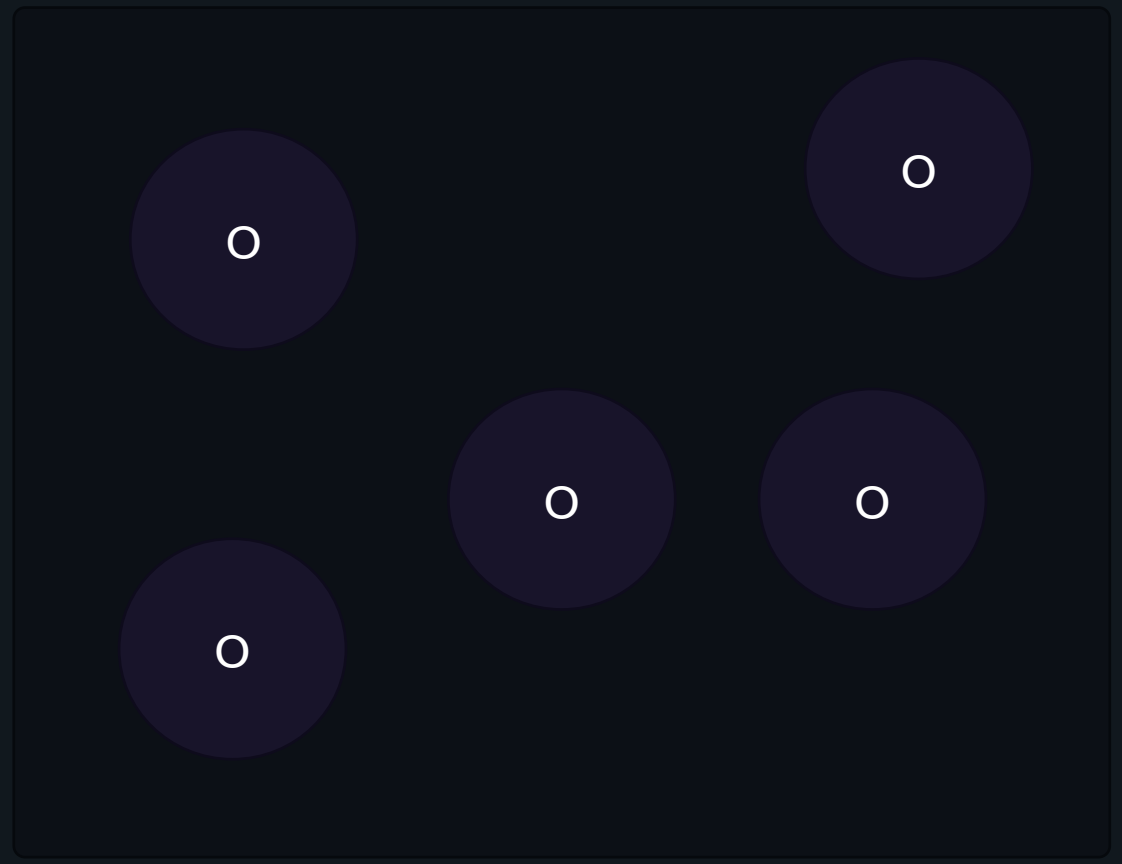
Anonymous go-routine

```
go func() {
  //code
}(args..)
```

KodeKloud

Go runtime scheduler

What happens when you start a Go program ?



Kernel space

Managed by the Kernel/Operating System

What happens when you start a Go program ?

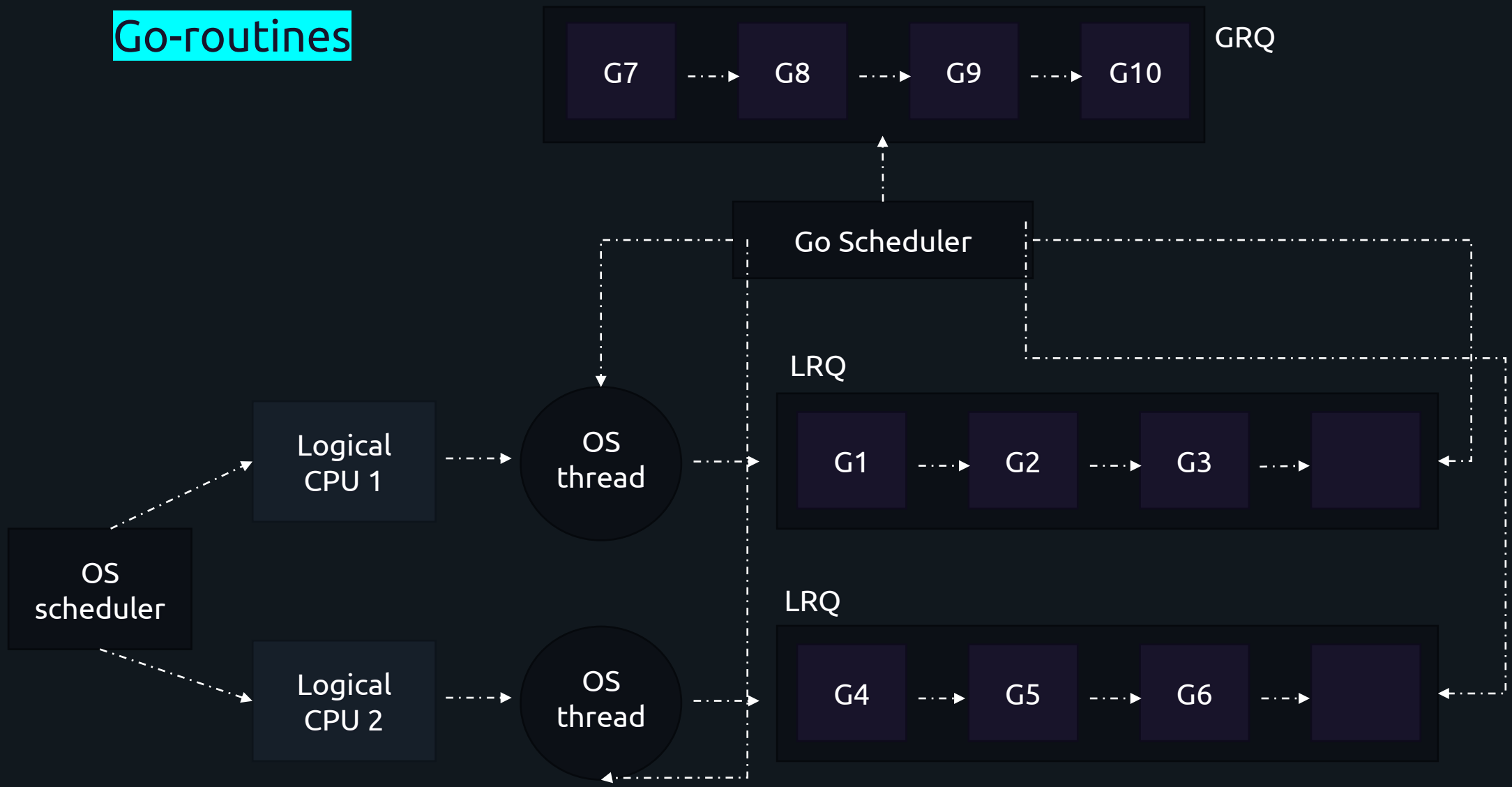
- We can find out the number of logical processors using the runtime.Numcpus method.

- Logical cores = number of physical cores * number of threads that can run on each core (hardware threads)

Go-routines

- Considered as a lightweight application-level thread that has a separate independent execution.
- The go runtime has its own scheduler that will multiplex the go-routines on the OS level threads in the go runtime.
- It schedules an arbitrary number of go-routines onto an arbitrary number of OS threads (m:n multiplexing).

Go-routines



KodeKloud

Cooperative Scheduler

- Golang scheduler is a cooperative scheduler.
- Cooperative scheduling is a style of scheduling in which the OS never interrupts a running process to initiate a context switch from one process to another.
- Processes must voluntarily yield control periodically or when logically blocked on a resource.
- Of course, there are some specific check points where go-routine can yield its execution to other go-routine. These are called context switches.

Context Switching

- Functions Call
- Garbage Collection
- Network Calls
- Channel operations
- On using go keyword

Go-routines vs Threads

- Go-routines are cheaper.
- Go-routines are multiplexed to a fewer number of OS threads.
- The context switching time of go-routines is much faster.
- Go-routines communicate using channels.

KodeKloud

Wait groups

Wait groups

- The problem with go-routines was the main go-routine terminating before the go-routines completed or even began their execution.
- To wait for multiple go-routines to finish, we can use a wait group.
- A wait group is a synchronization primitive that allows multiple go-routines to wait for each other.
- This package acts like a counter that blocks execution in a structured way until its internal counter becomes 0.

Wait groups - Syntax

```
import "sync"

var     wg   sync.WaitGroup
```

Wait groups - Methods

```
wg.Add(int)
```

This indicates the number of available go-routines to wait for.
The integer in the function parameter acts like a counter.

Wait groups - Methods

```
wg.Wait()
```

This method blocks the execution of code until the internal counter reduces to value = 0.

Wait groups - Methods

```
wg.Done()
```

- This method decreases the internal count parameter in Add() method by 1.

Wait groups

	Count = 0
wg.Add(3)	Count = 3
wg.Done()	Count = 2
wg.Done()	Count = 1
wg.Done()	Count = 0

KodeKloud

KodeKloud

Channels

Channels

- Channels are a means through which different go-routines communicate.
- “Do not communicate by sharing memory; instead, share memory by communicating” – Rob Pike
- Communicate by sharing memory – Threads and mutexes.
- Share memory by communicating – Go-routines and channels.

Channels

- The communication is bidirectional by default, meaning that you can send and receive values from the same channel.
- By default, channels send and receive until the other side is ready.
- This allows go-routines to synchronize without explicit locks or condition variables.

Syntax

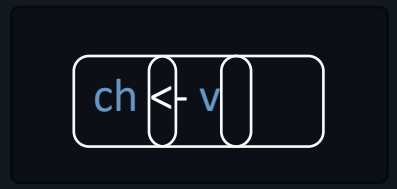
```
var c chan string
```

```
c := make( chan string)
```

Channel Operations

- Sending a value
- Receiving a value
- Closing a channel
- Querying Buffer of a channel
- Querying length of a channel

Channel Operations: Sending a value



- <- is a channel send operator.
- This operator is used to send a value to the channel.
- v must be a value which is assignable to the element type of channel ch.

Channel Operations: Receiving a value

- <- is a channel receiving operator.
- This is used to receive a value from a channel.
- val is a variable in which the read data from the channel will be stored.

Channel Operations: Closing a channel

```
close(ch)
```

- close() is a built-in function.
- The argument of a close function call must be a channel value.

Channel Operations: Querying buffer of a channel

```
cap(ch)
```

- cap() is a built-in function.
- returns an integer denoting the buffer of the specified channel.

Channel Operations: Querying length of a channel

```
len(ch)
```

- len() is a built-in function.
- returns an integer denoting the length of the specified channel.

KodeKloud

Buffered Channels

Unbuffered Channels

- A channel that needs a receiver as soon as a message is emitted to the channel.

- We do not declare any capacity, and it cannot store any data.

Buffered Channels

- Have some capacity to hold data.

- On a buffered channel
  - Sending to a channel, blocks the go-routine, only if the buffer is full.
  - Receiving from a channel blocks only when the channel is empty.

Syntax

```
c := make( chan <data_type, capacity)
```

```
c := make( chan int, 10)
```

Length of an unbuffered channel

- Builtin len() function can be used to get the length of a channel.
- The length of a channel is the number of elements that are already there in the channel.
- So, length represents the number of elements queued in the buffer of the channel.
- Length of a channel is always less than or equal to the capacity of the channel. (length<= capacity)
- Length of unbuffered channel is always zero.

KodeKloud

Closing a channel

Closing a channel

- Closing a channel means that no more data can be sent to that channel.
- It is generally closed when there's no more data to be sent.
- We can use the inbuilt close() function for the operation.

Closing a channel

```
v, ok := <- ch
```

- If ok is true, this means that the channel is open.
- If ok is false, this means that the channel is closed and there are no more values to receive.

Panic situations

- In Go language, panic is just like an exception, it also arises at runtime.
- Panic means an unexpected condition arises in your Go program due to which the execution of your program is terminated.
- There are a few scenarios that can cause panic while working with channels such as –
    - sending to a channel after it has been closed.
    - closing an already closed channel.

KodeKloud

select statement

select statement

- The *select* statement in Go looks like a switch statement but for channels.
- The *select* statement lets a go-routine wait on multiple communication operations.
- In *select*, each of the case statement waits for a send or receive operation from a channel.
- Whereas in *switch*, each of the case statements is an expression.

select statement

- *select* blocks until any of the case statements are ready.
- If multiple case statements are ready, then it selects one at random and proceeds.

select statement: syntax

```
select {

 case channel_send_or_receive:
   // Do something

 case channel_send_or_receive:
   // Do something

 default:
   // (optional)


}
```

select statement: applications

- The *select* statement lets a go-routine wait on multiple communication operations.

- *select* along with channels and go-routines becomes a very powerful tool for managing synchronization and concurrency.

select statement: default case

- Like *switch* statement, we can have a default case in *select* too.
- This default case will be executed if no send it or receive operation is ready on any of the case statements
- Default block makes the select non-blocking as default case will be executed if all the other cases are blocked.

select vs switch

- *switch* - Non-blocking.
- *select* - Statements can block since they are used with channels, and they can block or receive operation.
- *switch* - Deterministic and will run in sequence to select the matching case.
- *select* - Non-deterministic, as it will execute a case randomly with no sequence.

KodeKloud

Cleaning up go-routines

go-routine leak

- Whenever you launch a go-routine function, you must make sure that it will eventually exit.
- A go-routine that would never terminate, forever occupies the memory it has reserved. This kind of memory leak is called go-routine leak.
- Go-routines leak if they end up either blocked forever on I/O like channel communication or fall into infinite loops.

KodeKloud

Buffered channels

When to use buffered channels ?

- Proper use of buffered channel means that you must handle the case where the channel is blocking, which might happen due to waiting on sender/receiver.

- Buffered channels are useful when you know how many go-routines you have launched, want to limit the number of go-routines you will launch, or want to limit the amount of work that is queued up.

KodeKloud

Time out code

Time out code

- Most interactive programs must return a response within a certain amount of time.
- Blocking timeout in select can be achieved by using *After* function of time package.

Time out code

```
func After(d Duration) <- chan Time
```

- The After function waits for d duration to finish and then it returns the current time on a channel.

KodeKloud

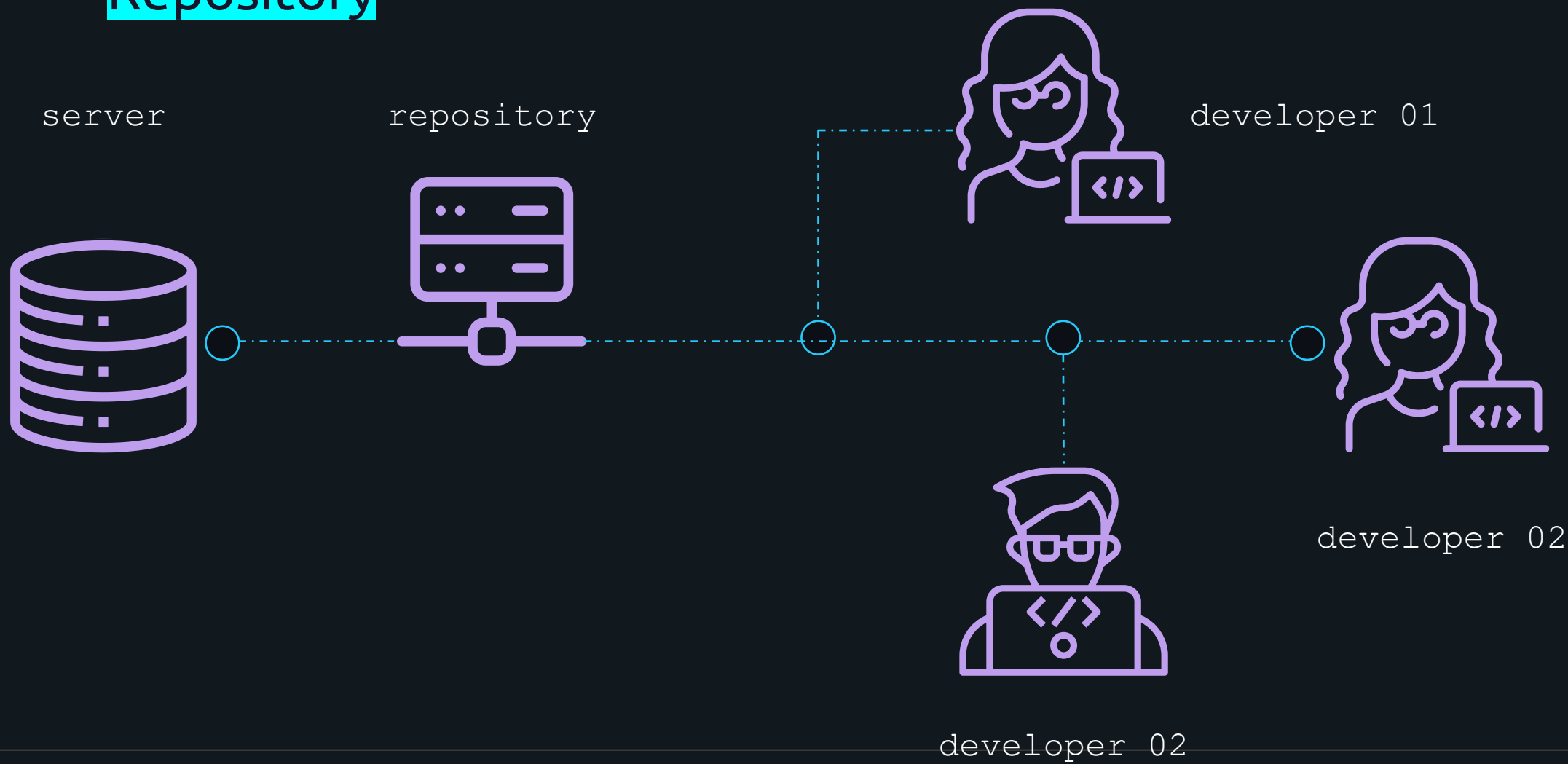
Modules and Packages

Repository

Modules

Packages

Repository



KodeKloud

Module & Packages

Globally unique identifier

github.com/kodekloud/learn

example.com/learn

...

Module & Packages

Go code is grouped into packages, and packages are grouped into modules.

A module specifies the dependencies needed to run our code, including the Go version and the set of other modules it requires in the go.mod file.

KodeKloud

go.mod file

go.mod file

A collection of Go source code becomes a module when there's a valid *go.mod* file in its root directory.

It consists of the module declaration, the minimum compatible version for Go, and the dependencies for the imported third-party packages.

KodeKloud

go commands

go mod init

- The command initializes and writes a new go.mod file in the current directory.
- in effect creating a new module rooted at the current directory.
- It accepts one optional argument, which is the module path for the new module.

go mod tidy

- The command 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.

go run <file>

- The command compiles and runs the program.

- Internally it
  - compiles your code and builds an executable binary in a temporary location,
  - launches that temp exe-file and
  - finally cleans it when your app exits.

- This command is useful for testing small programs during the initial development stage of your application.

go build

- The go build command compiles the packages named by the import paths, along with their dependencies into an executable.

- The executable gets created in the current source directory.

go install

- go build will compile and create the executable in current directory,

- go install will
  - compile and
  - move the executable to $GOPATH/bin
  - run this executable from any path on the terminal.

- go env GOPATH

go get

- The command
  - resolves its command-line arguments to packages at specific module versions,
  - updates go.mod to require those versions,
  - and downloads source code into the module cache.

- Add a dependency for a package or upgrade it to its latest version: `go get example.com/pkg`

- To upgrade or downgrade a package to a specific version:
  `go get example.com/pkg@v1.2.3`

KodeKloud

Developing and publishing a module

Workflow

- Design and code the packages that the module will include.
- Commit code to your repository using conventions that ensure it’s available to others via Go tools.
- Publish the module to make it discoverable by developers.
- Over time, revise the module with versions that use a version numbering convention that signals each version’s stability and backward compatibility.

Design and Development

- When you're designing a module's public API, try to keep its functionality focused and discrete.
- Ensure backward compatibility.

Decentralized publishing

- In Go, you publish your module by tagging its code in your repository to make it available for other developers to use.
- Developers use Go tools (including the go get command) to download your module's code to compile with.

Package discovery

- After you've published your module and someone has fetched it with Go tools, it will become visible on the Go package discovery site at pkg.go.dev.

- There, developers can search the site to find it and read its documentation.

Versioning

- As you revise and improve your module over time, you assign version numbers
  - designed to signal each version's stability and backward compatibility

- You indicate a module's version number by tagging the module's source in the repository with the number.

KodeKloud

Module version numbering

Module version numbering

- A module's developer uses each part of a module's version number to signal the version's stability and backward compatibility.

- For each new release, a module's release version number specifically reflects the nature of the module's changes since the preceding release.

Module version numbering

V 1.4.0 – beta.2

Module version numbering

V 1.4.0 – beta.2

major version

Module version numbering

V 1.4.0 – beta.2

minor version

Module version numbering

V 1.4.0 – beta.2

patch version

Module version numbering

V 1.4.0 – beta.2

Pre-release identifier

KodeKloud

godoc

godoc

- Godoc parses Go source code - including comments - and produces documentation as HTML or plain text.

- The result is documentation tightly coupled with the code it documents.

godoc

- The convention is simple: to document a type, variable, constant, function, or even a package
    - write a regular comment directly preceding its declaration, with no intervening blank line.
    - Use a blank comment to break your comment into multiple paragraphs.

- If you have lengthy comments for the package), the convention is to put the comments in a file in your package called doc.go.

godoc

```
go doc PACKAGE_NAME
```

```
go doc PACKAGE_NAME.IDENTIFIER_NAME
```

KodeKloud

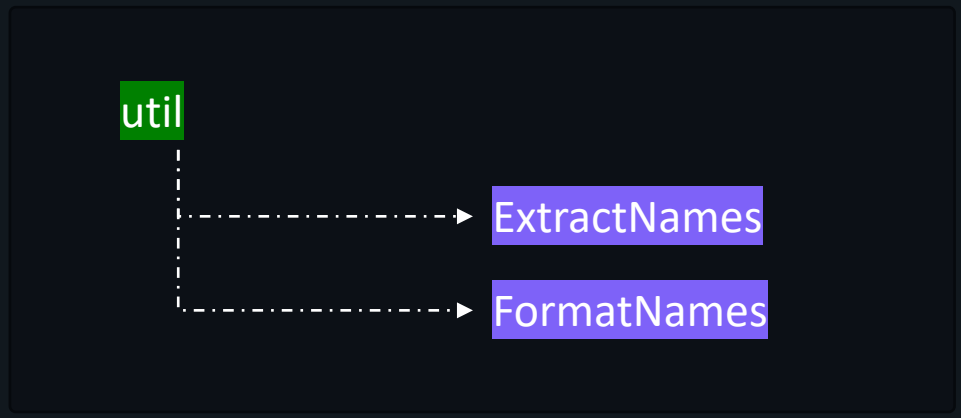
Naming packages

Naming packages

- Package names should be descriptive.

Naming packages

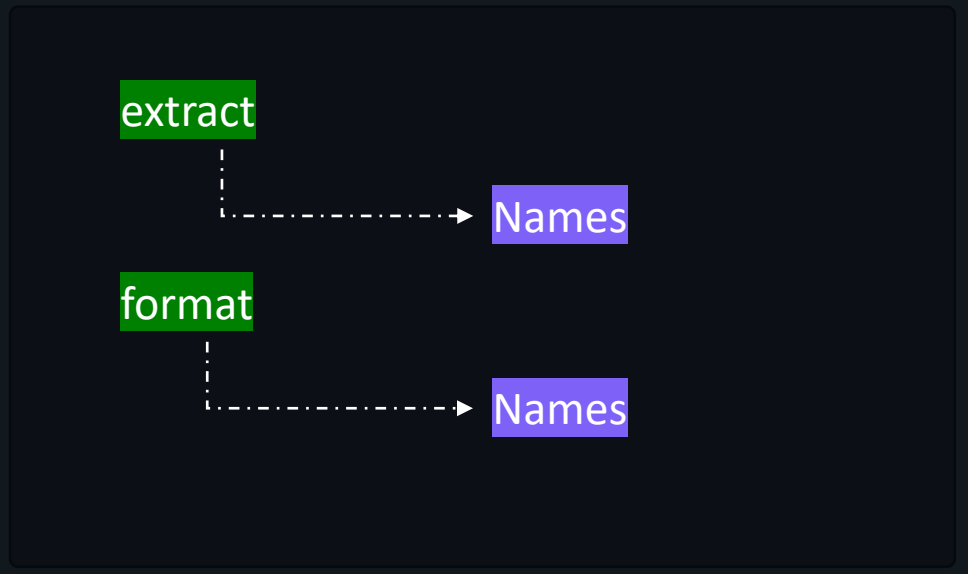
- extract names from a string.
- format a string.



- util.ExtractNames
- util.FormatNames

Naming packages

- extract names from a string.
- format a string.



- extract.Names
- format.Names

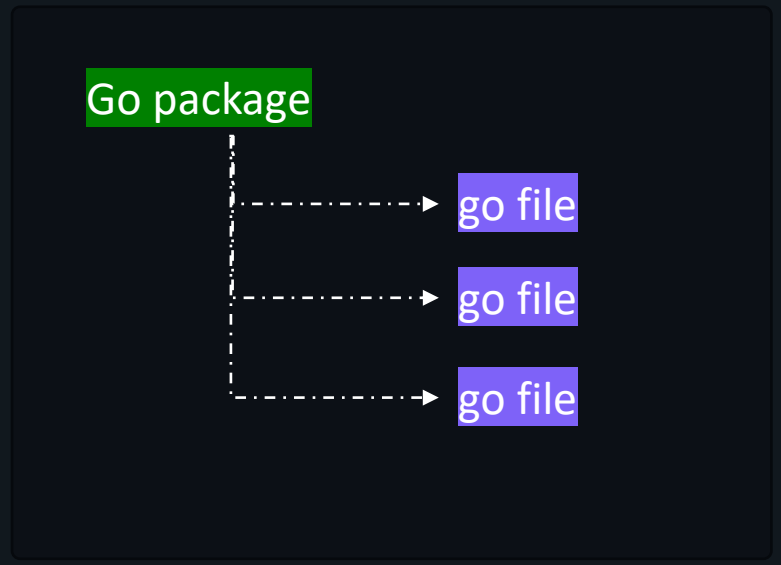
Naming packages

- Package names should be descriptive.
- Avoid repeating the name of the package in the names of functions and types within the package.
- Every Go file in a directory must have an identical package clause.
- As a rule, you should make the name of the package match the name of the directory that contains the package.

KodeKloud

Core Packages

Core packages



KodeKloud

Core packages

Core packages

- When a code is written is by some one else and comes bundled in a package, we refer to it as third-party package, this is usually added as a dependency on your project.

- When packages come as a part of the offerings by the programming language itself. You do not need to add any third-party dependency on your project to avail these.

Core packages

- In Go, these packages come bundled together as a library. This library is called the “Standard Library”.

- Each of these packages contains code that implements a functionality.

Core packages

- trings
- input/output
- File handling
- errors
- hashes and cryptography
- sort
- testing

KodeKloud

Strings

Contains

- Used to search whether a particular text/string/character is present in a given string.

- If the text/string/character is present in the given string, then it returns True, else it returns False.

Contains

ReplaceAll

- The ReplaceAll function in strings package can be used to replace any text with a given text.

ReplaceAll

```
func ReplaceAll(str, old, new string) string
```

Count

- The Count function counts the number of times a word occurs in a given sentence.

Count

```
func Count(first, second string) int
```

KodeKloud

Input/Output

Input/Output

- From writing and reading a file, to writing and reading an HTTP response, to processing database operations, all processes are based on I/O.

- The io standard library aims to abstract these existing processes and tasks, which appear in countless places, into a shared interface.

io interfaces

- The Reader interface
- The Writer interface

io Reader interface

- The Reader interface basically provides the Input function.

```
type Reader interface {
        Read(p [] byte) (n int, err error)
}
```

io Writer interface

- The Writer interface basically provides a generic way to output data.

```
type Writer interface {
        Write(p [] byte) (n int, err error)
}
```

KodeKloud

File handling

File handling

- In programming, file handling essentially means working with it, such as getting metadata information, creating new files, or simply reading and writing data to and from a file.

- In Golang, the API for file handling is well-knitted into the standard architecture and we can do it using the standard libraries.

File handling libraries

- The os package provides an API interface for file handling which is uniform across all operating systems.
- It provides functionality such as creation, deletion, opening a file, modifying its permissions and so on.
- The io package provides interfaces for basic I/O primitives and wraps them into easy-to-use public interfaces.
- The filepath package that would provide functions to parse and construct file paths.
- We also use the fmt package to format I/O with functions to read and write to the standard input and output.

Constructing file paths

- Using the filepath package
- It provides functions to parse and construct file paths in a way that is portable between operating systems
- dir/file on Linux vs. dir\file on Windows, for example.

Constructing file paths

Summary

- Join method that constructs paths in a portable way.
- Dir and Base method that split a path to the directory and the file.
- IsAbs method to check if a path is absolute or not.
- Ext method to get the extension from a filename.

Appending to a file

- We will use its built-in package called os with a method OpenFile() to append text to a file.

KodeKloud

Errors

Errors

- New() function
- Errorf() function

KodeKloud

Logging

Why to Log ?

- Spot bugs in the application's code
- Discover performance problems
- Do post-mortem analysis of outages and security incidents

What to Log ?

- The timestamp for when an event occurred, or a log was generated
- Log levels such as debug, error, or info
- Contextual data.

log package

- The Go standard library has a built-in log package that provides most basic logging features.

- While it does not have log levels (such as debug, warning, or error), it still provides everything you need to get a basic logging strategy set up.

log package

- The Go standard library has a built-in log package that provides most basic logging features.

- While it does not have log levels (such as debug, warning, or error), it still provides everything you need to get a basic logging strategy set up.

- Great for local development when getting fast feedback is more important than generating rich, structured logs.

Logging frameworks.

- Logging framework helps in standardizing the log data.
- It's easier to read and understand the log data.
- glog and logrus
- logrus is better maintained and used in popular projects like Docker.

KodeKloud

Hashes and Cryptography

Crypto package

- Cryptography is the practice of ensuring secure communication in the presence of third parties.
- It makes use of several encryption, decryption and cryptic techniques to ensure that digital data is not exploited by unwanted and harmful entities.
- Go provides good support for cryptography and hashing through the package **crypto**.

crypto package

- aes
- cipher
- rsa
- sha

KodeKloud

Testing

Testing package

- Go has a built-in test runner and framework for standard language tooling.
- The easiest way to run unit tests for your project is using a command like `go test ./...` from the command line.
- Running this command will discover and run any tests within your current directory or their subdirectories.
- Tests must be in files separate from your main package code and end with the suffix `_test.go`.

KodeKloud

Web Server and API

Web server

- The primary role of a web server is to serve web pages for a website.
- If you want to host your web application on the internet, in many cases you will need a web server.
- A web page can be rendered from a single HTML file, or a complex assortment of resources fitted together. (such as JS, scripts, CSS files etc.)

Request-Response model

- the client sends a request for some data
- and the server responds to the request.

Request-Response model

use cases for a web server

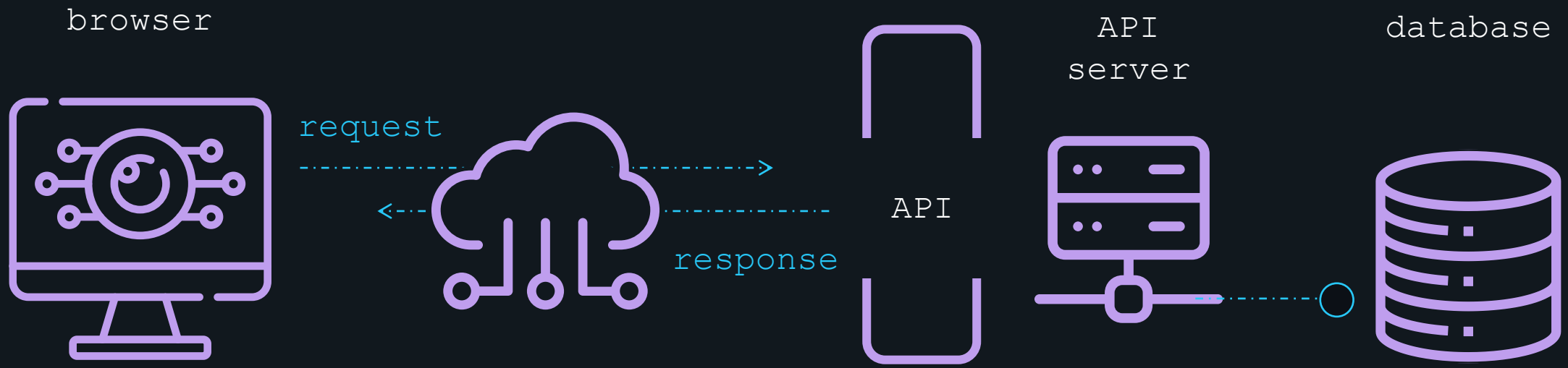
- Serves HTML, CSS and JS files.
- Serves images and videos.
- Handles HTTP error messaging
- Handles user requests, often concurrently.
- Directs URL matching and rewriting.
- Processes and serves dynamic content.

API

- API stands for application programming interface, which is a set of definitions and protocols for building and integrating application software.

- APIs let your product or service communicate with other products and services without having to know how they're implemented.

API



KodeKloud

HTTP Verbs

HTTP verbs

- HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

- Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs.

GET

- The GET method requests a representation of the specified resource.
- Requests using GET should only retrieve data.

POST

- The POST method submits an entity to the specified resource, often causing a change in state or side effects on the server.

PUT

- The PUT method replaces all current representations of the target resource with the request payload.

DELETE

- The DELETE method deletes the specified resource.

KodeKloud

REST

REST

- REST stands for Representational State Transfer..
- REST is a set of architectural constraints, not a protocol or a standard.
- When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint.
- This data is delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text.
- JSON is the most generally popular file format to use.

REST

- A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
- Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
- A uniform interface between components so that information is transferred in a standard form.
- Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

KodeKloud

Project Explanation

Product table

Id	name	quantity	price

API endpoints

GET	/products	Gets a list of all the products

API endpoints

GET	/products	Gets a list of all the products
GET	/product/id	gets the information about the respective product

API endpoints

GET	/products	Gets a list of all the products
GET	/product/id	gets the information about the respective product
POST	/product	creates a new product based on the given information from the user and saves it to the database

API endpoints

GET	/products	Gets a list of all the products
GET	/product/id	gets the information about the respective product
POST	/product	creates a new product based on the given information from the user and saves it to the database
PUT	/product/id	updates the respective product with the given information from the user

API endpoints

GET	/products	Gets a list of all the products
GET	/product/id	gets the information about the respective product
POST	/product	creates a new product based on the given information from the user and saves it to the database
PUT	/product/id	updates the respective product with the given information from the user
DELETE	/product/id	deletes the respective product

KodeKloud

gorilla/mux router

- Package gorilla/mux implements a request router and dispatcher for matching incoming requests to their respective handler.
- The name mux stands for "HTTP request multiplexer"
- Like the standard http.ServeMux,
  - mux.Router matches incoming requests against a list of registered routes and calls a handler for the route that matches the URL or other conditions.

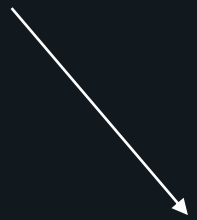
supports method-based routing

- router makes it easy to dispatch a HTTP request to different handlers based on the request method such as

- GET
- POST
- PUT
- DELETE
- PATCH

supports variables in URL paths

/movies/{id}



id is a dynamic value in the URL path