Go

Go Develop in 2007 and open source in 2009.

**Tools:**

MongoDb:

**1] MongoDB** is a **NoSQL database** that stores data in a flexible, JSON-like format called **BSON** (Binary JSON).

2] Unlike traditional relational databases (like MySQL or PostgreSQL), MongoDB doesn't require a fixed schema, which makes it great for handling unstructured or semi-structured data.

Kubernates : YouTube link 🡪 <https://www.youtube.com/watch?v=TlHvYWVUZyc>

**Kubernetes** (often abbreviated as **K8s**) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications.

**🔧 Why Kubernetes is used:**

1. **Container Orchestration**: Automatically manages multiple containers across a cluster of machines.
2. **Scalability**: Can automatically scale applications up or down based on load.
3. **Self-healing**: Restarts failed containers, replaces them, and reschedules them when nodes die.
4. **Load balancing**: Distributes traffic across containers.
5. **Service Discovery**: Provides DNS-based service discovery so containers can talk to each other easily.
6. **Rolling updates & rollbacks**: Update your application without downtime.
7. **Secret and config management**: Manages sensitive data and configuration separately from the application code.

**🐹 Kubernetes and Go (Golang):**

Kubernetes is **written in Go**, and Go is heavily used in its ecosystem. Here's how Go fits in:

1. **Kubernetes client libraries**:
   * You can use Go's client-go library to interact with the Kubernetes API.
   * This is helpful if you're writing tools, operators, or controllers that manage resources in a K8s cluster.
2. **Custom Controllers/Operators**:
   * With Go, you can write your own custom logic to manage Kubernetes resources.
   * Often used in DevOps, CI/CD, and infrastructure automation.
3. **Cloud-native development**:
   * Go is the preferred language for many cloud-native tools (like Docker, Prometheus, Helm) that integrate with Kubernetes.

**🧠 Example use case in Go:**

Let's say you want to write a Go program that lists all Pods in a Kubernetes cluster:

go

CopyEdit

import (

"context"

"fmt"

"k8s.io/client-go/kubernetes"

"k8s.io/client-go/tools/clientcmd"

metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"

)

func main() {

config, \_ := clientcmd.BuildConfigFromFlags("", "/path/to/kubeconfig")

clientset, \_ := kubernetes.NewForConfig(config)

pods, \_ := clientset.CoreV1().Pods("default").List(context.TODO(), metav1.ListOptions{})

for \_, pod := range pods.Items {

fmt.Println(pod.Name)

}

}

📝 This is just a basic snippet. You can use client-go for full-fledged automation, deployment tools, etc.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Jenkins \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Jenkins** is an **open-source automation server** used primarily for **continuous integration and continuous delivery/deployment (CI/CD)**. It helps developers **automate** the parts of software development related to building, testing, and deploying code.

# **What Jenkins Does:**

1. **Automates Builds**  
   It pulls code from version control systems (like Git), compiles it, and builds it into an executable.
2. **Runs Tests Automatically**  
   Jenkins can run automated tests every time you push new code to make sure nothing breaks.
3. **Deployment**  
   It can automatically deploy your application to a server or cloud after a successful build.
4. **Saves Time**  
   Instead of doing things manually, Jenkins runs everything automatically, 24/7.

**# Why Jenkins is Used:**

* **CI/CD Pipelines**: Speeds up development by integrating and testing code automatically.
* **Open-Source and Free**
* **Extensible**: Has 1,800+ plugins for tools like Docker, GitHub, Slack, AWS, etc.
* **Easy Integration**: Works with most tools in the DevOps ecosystem.
* **Custom Pipelines**: You can write powerful pipelines as code using Groovy or YAML.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Visual Studio Code \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Code Editor

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***42 crunch*** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

To increase the code quality of swagger file.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Cloud*** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

AWS

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Go In Built Function*** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1] Make() 🡪 To allocate memory

2] reflect.DeepEqual 🡪 Used to compare two slices

Syntax 🡪 deepEqualResult := reflect.DeepEqual(slice1, slice2)

3] Sort 🡪 [ slice]

🡪 sort.Strings(slice\_name)

* sort.Ints(slice\_name)

2]sort.IntsAreSorted(intSlice) 🡪 for checking slice is sorted or not.

3] *// Sort in descending order*

sort.Slice(intSlice, **func**(i, j int) bool {

**return** intSlice[i] > intSlice[j]

})

4] strings

strings.Count 🡪 count substring from original string and return.

strings.Fields 🡪 is a function from the strings package that splits a string into a slice of substrings based on whitespace characters (spaces, tabs, newlines, etc.). It treats any whitespace as a delimiter and removes extra whitespace.

strings.Replaceall 🡪

strings.Split :

In Go, the strings.Split(str1, "") function is used to split a string (str1) into a slice of its individual characters.

**Explanation:**

* strings.Split splits a string into substrings based on a separator.
* When the separator is an empty string (""), it causes the string to be split between each individual character.

This means that strings.Split(str1, "") will return a slice of strings, where each string is a single character from the original string.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Files \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**In a Go (Golang) project, there are several key files and directories typically used to structure the project. These files help manage dependencies, configuration, and project setup. Here's a breakdown of the most common files in a Go project, what they contain, and why they are used:**

**1. main.go (or other Go source files)**

* **Purpose:** This is the entry point of a Go application. It contains the main function (func main()) that is executed when the program runs.
* **Content:**
  + The main.go file typically contains the program’s logic and any initial setup (e.g., initializing configurations, setting up routes, etc.).
  + Other Go files like handlers.go, models.go, utils.go are also used to break up code into manageable parts, but main.go will often initialize everything.
* Example:

go

CopyEdit

package main

import "fmt"

func main() {

fmt.Println("Hello, Go!")

}

* **Why it's used**: It is the entry point of your program, and it ties everything together. Every Go application typically starts in the main package.

**2. go.mod (Go Modules)**

* Purpose: The go.mod file is used to manage the dependencies of the project in Go modules. It defines the module's path and the Go version the project is using.
* **Content:**
  + Module path: Identifies the project (like a URL).
  + Dependencies: Lists external libraries or packages required by your project.
* **Example:**

module github.com/username/myproject

go 1.18

require (

github.com/gin-gonic/gin v1.7.4

github.com/stretchr/testify v1.7.0

)

* **Why it's used**: It helps manage dependencies and versions. Since Go 1.11, Go Modules has been the standard for dependency management, and the go.mod file is automatically generated when running go mod init.

**3. go.sum (Checksum file for modules)**

* Purpose: The go.sum file is used to store cryptographic hashes of dependencies listed in the go.mod file.
* Content:
  + It contains checksums for the dependencies, ensuring that they are verified for consistency and integrity.
* Example:

text

github.com/gin-gonic/gin v1.7.4 h1:1234567890abcdef1234567890abcdef12345678

* **Why it's used:** It provides additional security and ensures that the dependencies are verified and have not been tampered with. The go.sum file is automatically updated when you add or update dependencies.

**4. README.md**

* **Purpose:** This file provides an overview of the project. It usually contains instructions on how to set up, use, and contribute to the project.
* **Content:**
  + Project description
  + Setup and installation instructions
  + Usage examples
  + Contribution guidelines
  + License information
* **Why it's used:** It's essential for other developers who might want to use or contribute to the project. It gives a high-level understanding of what the project is about.

**5. Makefile (optional, for automation)**

* Purpose: A Makefile is often used to automate common tasks like building the project, running tests, or deploying the application. It's not native to Go but is widely used in many programming projects.
* **Content:**
  + A list of targets (commands) such as build, test, lint, etc.
* Example:

makefile

CopyEdit

build:

go build -o myapp main.go

test:

go test ./...

lint:

golangci-lint run

* **Why it's used:** It simplifies repetitive tasks and can help developers automate common actions with a single command (e.g., make build).

**6. cmd/ (optional)**

* Purpose: This directory is used to store the entry points of your application, typically one for each command your application can execute.
* **Content:**
  + Inside the cmd/ folder, you'll often find a subdirectory for each executable, each containing a main.go file.
* Example:

text

CopyEdit

cmd/

myapp/

main.go

myworker/

main.go

* **Why it's used**: It's a good organizational pattern when your project has multiple executable applications or microservices. It helps keep the entry points organized and separate.

**7. pkg/ (optional)**

* **Purpose:** This folder contains libraries or utility functions that are shared across the project. It’s often used for reusable packages that can be imported in other parts of the project**.**
* **Content:**
  + Go files that define common functionality like helpers, services, or data models.
* **Example:**

text

CopyEdit

pkg/

util/

util.go

service/

service.go

* **Why it's used:** It helps in modularizing the code and organizing common functionality that can be reused in various parts of the project.

**8. internal/ (optional, for internal packages)**

* **Purpose**: The internal/ directory is used for packages that are meant to be used only within the project. These packages cannot be imported by other projects.
* **Content:**
  + Go packages that should remain internal to the project, preventing external use.
* **Example:**

**text**

CopyEdit

internal/

db/

db.go

* **Why it's used:** Go enforces that packages inside internal/ can only be imported within the project, preventing external code from using sensitive or project-specific functionality.

**9. vendor/ (optional)**

* **Purpose:** The vendor/ directory contains copies of all the dependencies used by the project. It’s created when you run go mod vendor.
* **Content:**
  + The entire dependency tree for your project.
* **Why it's used:** It allows the project to work offline and ensures that all dependencies are included in the repository. This is optional when using Go Modules but may be useful for specific situations (e.g., CI/CD pipelines or internal repositories).

**10. config/ (optional)**

* **Purpose**: This directory stores configuration files, such as environment variable configurations or JSON/YAML files that define app settings.
* **Content:**
  + Configuration files (e.g., config.json, config.yaml, .env).
* **Example:**

**text**

**CopyEdit**

**config/**

**config.yaml**

**secrets.env**

* **Why it's used: It keeps configuration and secret management organized and separate from the main application code.**

**Summary**

* **main.go: Entry point for the application.**
* **go.mod: Manages dependencies.**
* **go.sum: Stores cryptographic hashes for dependencies.**
* **README.md: Project overview and instructions.**
* **Makefile: Automation for common tasks.**
* **cmd/: Entry points for different executable applications.**
* **pkg/: Reusable libraries or utilities.**
* **internal/: Packages meant for internal use only.**
* **vendor/: Copies of project dependencies (optional).**
* **config/: Stores configuration files.**

**These files and directories help organize a Go project, improve code maintainability, and make it easier for other developers to contribute to the project.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Packages \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**In a Go project, packages play a crucial role in organizing and managing code into reusable modules. Go's simplicity and modularity allow developers to break up a project into multiple packages that can be used across the application.**

**Here are the common Go packages used in a typical Go project, along with their uses and what they help you achieve:**

**1. main Package**

* **Use: The main package is special in Go and serves as the entry point of your application. The main package contains the main() function, which is executed when the application starts.**
* **Required for: Any executable Go application.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import "fmt"**

**func main() {**

**fmt.Println("Hello, Go!")**

**}**

* **Why it's used: It's essential for the application to run. The Go runtime looks for the main() function inside the main package when starting the program.**

**2. fmt Package**

* **Use: The fmt package is used for formatted I/O, such as printing to the console or reading input. It's one of the most common packages in Go.**
* **Required for: Printing output to the terminal, formatted strings, etc.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import "fmt"**

**func main() {**

**fmt.Println("Hello, Go!")**

**}**

* **Why it's used: It provides functions like fmt.Println() for printing and fmt.Sprintf() for string formatting, making it easy to work with text output.**

**3. os Package**

* **Use: The os package allows interaction with the operating system, such as handling file systems, environment variables, and process-related tasks.**
* **Required for: Handling files, environment variables, and interacting with the system at a lower level.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import (**

**"fmt"**

**"os"**

**)**

**func main() {**

**args := os.Args**

**fmt.Println("Arguments:", args)**

**}**

* **Why it's used: It provides functions to interact with the operating system, such as working with command-line arguments, files, and directories.**

**4. net/http Package**

* **Use: The net/http package provides HTTP client and server functionalities. It's used for building web servers or making HTTP requests.**
* **Required for: Web servers, REST API servers, making HTTP requests.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import (**

**"fmt"**

**"net/http"**

**)**

**func main() {**

**http.HandleFunc("/", func(w http.ResponseWriter, r \*http.Request) {**

**fmt.Fprintf(w, "Hello, Go HTTP Server!")**

**})**

**http.ListenAndServe(":8080", nil)**

**}**

* **Why it's used: It's critical for building web-based applications, whether you're creating a simple server or handling HTTP requests and responses.**

**5. time Package**

* **Use: The time package is used for time-related functionalities, like getting the current time, parsing dates, formatting dates, and working with durations.**
* **Required for: Managing time, delays, and time-based logic in the application.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import (**

**"fmt"**

**"time"**

**)**

**func main() {**

**currentTime := time.Now()**

**fmt.Println("Current Time:", currentTime)**

**}**

* **Why it's used: Time manipulation is often needed in applications (e.g., scheduling tasks, logging, or displaying timestamps).**

**6. log Package**

* **Use:** The log package is used for logging messages, including error handling. It helps track the behavior of an application and debug problems.
* **Required for:** Logging messages, errors, and application state.
* **Example:**

**go**

**package main**

**import (**

**"log"**

**"os"**

**)**

**func main() {**

**log.SetOutput(os.Stdout)**

**log.Println("This is a log message")**

**}**

* **Why it's used: It's useful for logging messages, errors, and debugging, which is essential for production applications.**

**7. errors Package**

* **Use:** The errors package is used to create error values. It’s useful for defining custom error messages and handling errors in the application.
* **Required for:** Error handling and generating custom error messages.
* **Example:**

**go**

**package main**

**import (**

**"errors"**

**"fmt"**

**)**

**func main() {**

**err := errors.New("This is a custom error")**

**fmt.Println(err)**

**}**

* **Why it's used:** It helps in generating error messages to indicate that something went wrong, which is important for fault-tolerant applications.

**8. math Package**

* **Use:** The math package contains mathematical constants and functions like trigonometry, logarithms, and rounding.
* **Required for**: Performing mathematical operations.
* **Example:**

**go**

**package main**

**import (**

**"fmt"**

**"math"**

**)**

**func main() {**

**fmt.Println("Pi:", math.Pi)**

**fmt.Println("Sine of Pi:", math.Sin(math.Pi))**

**}**

* **Why it's used**: It's essential when performing mathematical computations, whether it's for simple math or complex calculations.

**9. strconv Package**

* **Use:** The strconv package is used for converting strings to other basic data types (like integers or floats) and vice versa.
* **Required for:** Converting between strings and numbers (parsing, formatting).
* **Example:**

**go**

**package main**

**import (**

**"fmt"**

**"strconv"**

**)**

**func main() {**

**str := "42"**

**num, \_ := strconv.Atoi(str) // Convert string to integer**

**fmt.Println("Converted Number:", num)**

**}**

* **Why it's used: It's very common to convert strings to numbers when dealing with user input or when reading from files or APIs.**

**10. encoding/json Package**

* **Use: The encoding/json package is used for encoding and decoding JSON data. It’s essential for working with APIs, data exchange, and configuration files.**
* **Required for: Working with JSON (parsing, encoding, and decoding).**
* **Example:**

**go**

**package main**

**import (**

**"encoding/json"**

**"fmt"**

**)**

**type Person struct {**

**Name string `json:"name"`**

**Age int `json:"age"`**

**}**

**func main() {**

**person := Person{Name: "John", Age: 30}**

**data, \_ := json.Marshal(person)**

**fmt.Println(string(data)) // Output: {"name":"John","age":30}**

**}**

* **Why it's used: JSON is a widely used data format for APIs and configuration files, so this package is key for most modern applications.**

**11. context Package**

* **Use: The context package is used to manage context across API boundaries, especially for cancellation signals, deadlines, and request-scoped values. It’s often used in concurrent programming.**
* **Required for: Managing timeouts, cancellation signals, and passing data across goroutines.**
* **Example:**

**go**

**CopyEdit**

**package main**

**import (**

**"context"**

**"fmt"**

**)**

**func main() {**

**ctx := context.Background()**

**fmt.Println(ctx)**

**}**

* **Why it's used: It's essential for managing timeouts, request cancellations, and scoped data (e.g., HTTP requests or database transactions).**

**Summary of Common Go Packages:**

| **Package** | **Use Case** |
| --- | --- |
| **main** | **Entry point for the application (main() func)** |
| **fmt** | **Formatted I/O (printing, formatting)** |
| **os** | **Interacting with the operating system (files, environment)** |
| **net/http** | **HTTP server and client functionality** |
| **time** | **Working with time (date, duration, etc.)** |
| **log** | **Logging messages and errors** |
| **errors** | **Creating and handling errors** |
| **math** | **Performing mathematical functions** |
| **strconv** | **Converting strings to and from other data types** |
| **encoding/json** | **Encoding and decoding JSON** |
| **context** | **Managing cancellation, timeouts, and scoped values** |

**These packages provide a solid foundation for most Go applications, whether you're building a command-line tool, web server, or performing mathematical operations. They help manage system interaction, data encoding, time management, and error handling.**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* GO \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Identifiers:**

are the user-defined names of the program components that used for identification purposes.

An identifier can be a variable name, function name, constant, statement label, package name, or type.

**Keywords:**

are reserved words/pre-defined words that have special meaning in the language that represent some predefined actions. You **cannot** use them as identifiers (like variable names, function names, etc.).

**📘 Here are all 25 keywords in Go:**

**Data types:**

Data Types specify the type of data.

Data type is divided into four categories which are as follows:

1. Basic type: Numbers, strings, and Booleans come under this category.
2. Aggregate type: Array and structs come under this category.
3. Reference type: Pointers, slices, maps, functions, and channels come under this category.
4. Interface type

***How to declare:***

*Constants are declared like variables but in using a const keyword as a prefix to declare a constant with a specific type.*

***It cannot be declared using “:=” syntax.***

**Operators:**

are the foundation of any programming language. Operators allow us to perform different kinds of operations on operands.

operators Can be categorized based on their different functionality:

* [Arithmetic Operators](https://www.geeksforgeeks.org/go-operators/#Arithmetic%20Operators)
* [Relational Operators](https://www.geeksforgeeks.org/go-operators/#Relational%20Operators)
* [Logical Operators](https://www.geeksforgeeks.org/go-operators/#Logical%20Operators)
* [Bitwise Operators](https://www.geeksforgeeks.org/go-operators/#Bitwise%20Operators)
* [Assignment Operators](https://www.geeksforgeeks.org/go-operators/#Assignment%20Operators)
* [Misc Operators](https://www.geeksforgeeks.org/go-operators/#Misc%20Operators)

**Limitations of Variadic Functions**

* Variadic functions can only have one variadic parameter, and it must be the last parameter.
* You cannot have multiple variadic parameters in a single function definition.

***Y Go?***

1. Go is a statically typed language, which means that the type of a variable must be declared before it can be used.
2. Go has a built-in garbage collector that automatically frees up memory when it is no longer needed.
3. Go has **strong support for concurrency**, allowing developers to write efficient and scalable code for multicore and distributed systems.
4. Go has a minimalist syntax that is easy to learn and read.
5. Go has a fast compiler that generates code that is optimized for modern hardware architectures.
6. Go has a standard library that provides support for a wide range of functionality, including networking, encryption, and file handling.
7. Go has a growing community of developers and a vibrant ecosystem of third-party packages and tools.
8. Go is used by many well-known companies for building large-scale distributed systems and high-performance applications.
9. Overall, Go is a powerful and efficient programming language that is well-suited for building modern applications and distributed systems. Its strong support for concurrency and minimalist syntax make it an attractive choice for developers who want to build scalable and efficient applications.

**Advantages and Disadvantages of Go Language**

**Advantages:**

1. **Flexible**– It is concise, simple and easy to read.
2. **Concurrency**– It allows multiple process running simultaneously and effectively.
3. **Quick Outcome**– Its compilation time is very fast.
4. **Library-** It provides a rich standard library.
5. **Garbage collection**– It is a key feature of go. Go excels in giving a lot of control over memory allocation and has dramatically reduced latency in the most recent versions of the garbage collector.
6. It validates for the interface and type embedding.
7. **Concurrency**: Go provides excellent support for concurrency, making it easy to write code that can run multiple tasks simultaneously. This is achieved through Goroutines and Channels, which allow you to write code that can run multiple operations at the same time.
8. **Performance**: Go is designed to be fast and efficient, with a focus on performance and low memory usage. This makes it well-suited for building high-performance network services, as well as for solving complex computational problems.
9. **Simplicity:** Go has a straightforward syntax and a simple type system, making it easy to learn and use, even for people with no prior programming experience.
10. **Garbage Collection:** Go has built-in garbage collection, which automatically manages memory for you. This eliminates the need for manual memory management, reducing the likelihood of memory leaks and other bugs that can arise from manual memory management.
11. **Statically Typed:**Go is a statically typed language, which means that types are determined at compile time. This provides stronger type safety and makes it easier to catch type-related bugs before they occur.

**Disadvantages:**

1. It has no support for generics, even if there are many discussions about it.
2. The packages distributed with this programming language is quite useful but Go is not so object-oriented in the conventional sense.
3. There is absence of some libraries especially a UI tool kit.
4. Limited Object-Oriented Features: Go does not have full-fledged object-oriented features like inheritance and polymorphism. This can make it more difficult to write complex programs, especially for developers who are used to traditional object-oriented languages.
5. No Generics: Go does not have built-in support for generics, which makes it difficult to write reusable code.
6. Immature Standard Library: Go’s standard library is relatively new and still maturing, which can make it difficult to find the tools you need for a particular task.

Benefits of Go

1] Evolution of infrastructure. (Infrastructure changed a lot)

- Go has more scalable & distributed infrastructure.

- Dynamic

- More Capacity

So with improved infrastructure Go performance multiple task at a time .

This make application more faster and user friendly.

e.g

1] we can download application.

Same time uploading and navigating the application.

2] watching YouTube video and parallelly reading or wring comment for the video.

2] To overcome changes of multi-threading, like multiple user editing same document or multiple users booking ticket at the same time,it may cause issue.

So Go was designed to run on multiple cores and built to support concurrency.

Concurrency in Go is cheap and easy.

In short:

1] Go is multi-threading programming language.

2]

\*\*\*\*\*\*\*\* Characteristics of Go \*\*\*\*\*\*\*\*\*\*\*

1] Simple and readable syntax of a dynamically typed language like python.

2] Efficiency and safety of a lower-level, statically typed lang like C++.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Advantage of Go:

1] Go has simple syntax which make the code maintainable easy to learn, read and write code.

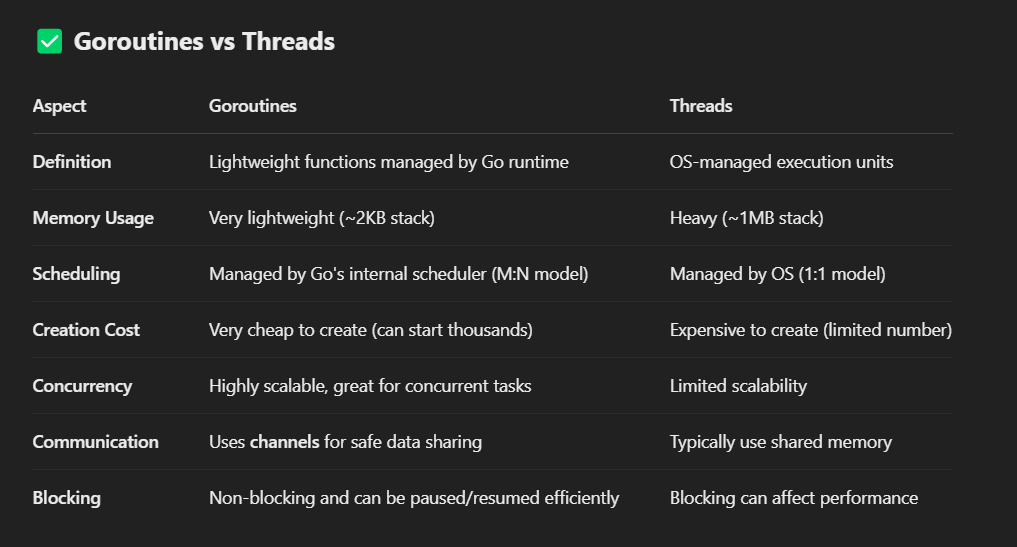
2] Go can build very Fast , there also very fast start up and run

3] It requires few resources. like cpu and run

4] Go code is transformed into machine code by compiler before it can run on a computer. So, it can deploy on different platform

Go convert in single binary form so they can deploy on different platform.

\*\*\* Difference \*\*\*



Garbage collector:

Go has a built-in, concurrent garbage collector that automatically frees memory no longer in use. It works using a mark-and-sweep algorithm and is optimized to reduce latency. This makes Go safe and efficient for high-performance backend services.

**⚙️ Key Features of Go's GC:**

| **Feature** | **Description** |
| --- | --- |
| Automatic | No need for manual memory management |
| Concurrent | Runs with your app to reduce pause time |
| Mark-and-Sweep | Traces reachable objects and frees unreferenced memory |
| Optimized for Low Latency | Designed for fast response times (e.g., <1ms pause) |
| GOGC | Environment variable to control GC frequency (GOGC=100 by default) |

**✅ Why Do We Use MongoDB?**

MongoDB is a **NoSQL database** that stores data in a **flexible, JSON-like format (BSON)** instead of traditional rows and tables like SQL databases.

**🔍 Key Reasons to Use MongoDB:**

**1. 🧾 Schema-less / Flexible Data Model**

* You don’t need to define tables or columns before inserting data.
* Documents in the same collection can have different fields.
* Great for **agile development** or **frequently changing data structures**.

**2. ⚡ High Performance**

* Optimized for **fast reads and writes**.
* Supports **horizontal scaling** with **sharding**.
* Ideal for **high-volume, low-latency** apps.

**3. 📦 Document-Oriented Storage**

* Stores data as **JSON-like documents** (BSON).
* Natural fit for **storing hierarchical or nested data** (like user profiles, product info, etc.).
* Easier mapping to **Golang structs**, **JavaScript objects**, etc.

**4. 🌐 Horizontal Scalability**

* Built-in **sharding** and **replication** allow it to scale out across multiple servers.
* Ensures **high availability** and **fault tolerance**.

**5. 🔄 Real-Time Analytics & Change Streams**

* Supports **real-time data streaming** via **change streams**.
* Good for building **event-driven systems** and **dashboards**.

**6. 🔐 Built-in Features**

* Indexing
* Aggregation framework (like SQL's GROUP BY, JOIN, etc.)
* Geospatial queries
* Full-text search
* Transactions (multi-document, since MongoDB 4.0)
* 3] interfaces and struct difference :

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **Aspect** | **Struct** | **Interface** |
| 1 | **Definition** | **Struct:** A composite data type that groups together fields (data). It is used to create concrete data types. | **Interface:** A type that defines a set of method signatures. It is used to specify behavior. |
| 2 | **Purpose** | Represents data/state. | Represents behavior/capabilities. |
| 3 | **Implementation** | Structs implement interfaces implicitly by defining the required methods. | Interfaces are satisfied by any type that has the required methods. |
| 4 | **Contains** | Fields and methods. | Only method signatures. |
| 5 | **Usage** | Used to model real-world entities with attributes. | Used to define behavior and enable polymorphism. |
| 6 | **Memory Type** | Value type with a defined memory layout. | Reference type internally storing type + value. |
| 7 | **Inheritance** | No inheritance; composition is preferred. | No explicit inheritance; interface satisfaction is implicit. |
| 8 | **Example** | type Person struct { Name string; Age int } | type Speaker interface { Speak() string } |

Compare this abv and below points and keep one.

**Key Differences**

1. **Composition vs. Method Definition**: Structs are about data composition, grouping related fields together. Interfaces are about method definition, specifying a set of method signatures without implementations.
2. **Implementation**: Structs can have concrete data and methods. Interfaces only define method signatures and do not contain any data.
3. **Usage**: Structs are used to model concrete things with specific properties. Interfaces are used to define capabilities or behaviors that can be implemented by different types.
4. **Polymorphism**: Interfaces enable polymorphism, allowing different structs to be treated uniformly based on shared behavior. Structs do not directly relate to polymorphism.

**When to Use?**

* Use **structs** when you need to group data and create concrete objects with specific properties.
* Use **interfaces** to define behaviors and capabilities, and when you need polymorphic behavior, allowing different structs to be treated uniformly based on shared behavior[3](https://www.bing.com/ck/a?!&&p=f027025b8fcc6e7b5ab6d6262eb973610888c38fa6fb2397c99aea271f658df1JmltdHM9MTc0NDc2MTYwMA&ptn=3&ver=2&hsh=4&fclid=321c66eb-79fe-61dd-2a51-73e4782b6069&u=a1aHR0cHM6Ly9kZXYudG8vcm9tdWxvZ2F0dG8vc3RydWN0cy1hbmQtaW50ZXJmYWNlcy1pbi1nby01MmNp&ntb=1).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Interface: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |
| --- | --- |
| What? | A type that defines method signatures |
| Why? | To enable 1**] abstraction**[You can write code that works with any type that implements a certain behaviour, **without knowing the concrete type**.]  **2] flexibility** [ Interfaces let you write flexible code that can work with **different types** using the **same interface**.]  **3] Decoupling**[Interfaces help to **decouple** code. That means different parts of your code don’t need to know about each other’s concrete types.  This is **super helpful in large projects**, especially when testing or working with external packages.  ]  4] **Mocking for Tests**  🡪 Interfaces are **great for testing** because you can easily create mock implementations.  🡪In tests, you can pass a fake DB that satisfies this interface instead of connecting to a real one. |
| How? | Any type that implements the methods is considered to satisfy the interface |
|  |  |

$$$$$$$$$$$$$$$$ Go dip $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

**1. Abstraction**

You can write code that works with any type that implements a certain behavior, **without knowing the concrete type**.

go

CopyEdit

func makeSound(a Animal) {

fmt.Println(a.Speak())

}

This function works with **any type** that has a Speak() method—like Dog, Cat, etc.

**✅ 2. Polymorphism**

Interfaces let you write flexible code that can work with **different types** using the **same interface**.

go

CopyEdit

type Dog struct{}

func (d Dog) Speak() string { return "Woof" }

type Cat struct{}

func (c Cat) Speak() string { return "Meow" }

makeSound(Dog{}) // Woof

makeSound(Cat{}) // Meow

**✅ 3. Decoupling**

Interfaces help to **decouple** code. That means different parts of your code don’t need to know about each other’s concrete types.

This is **super helpful in large projects**, especially when testing or working with external packages.

go

CopyEdit

type DB interface {

Save(data string) error

}

Now your application can work with **any database** (MySQL, PostgreSQL, in-memory) that implements Save().

**✅ 4. Mocking for Unit Tests**

Interfaces make it easy to create **mock objects** for testing.

For example, you can pass a mock DB implementation that pretends to save data, without hitting a real database.

**🔧 Real Life Analogy**

Think of an **interface like a remote control**:

* The remote (interface) says: “I need a button that says Play()”.
* Any device (TV, DVD, Music system) that has a Play() function can be controlled by this remote.
* The remote doesn't care *how* the device implements Play() — just that it exists.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Method*** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Function – A standalone piece of code:**

A **function** in Go is a block of code that performs a task, but it is **not associated with any type**.

func greet(name string) string {

return "Hello, " + name

}

You can call it like:

greet("Alice")

**🔸 Method – A function associated with a type:**

A **method** is just a function that has a **receiver** — it is tied to a specific **type** (like a struct).

type Person struct {

Name string

}

func (p Person) Greet() string {

return "Hello, " + p.Name

}

You call it like:

p := Person{Name: "Alice"}

p.Greet() // "Hello, Alice"

**🎯 So why use methods?**

**✅ 1. Organizing behavior with data**

* Methods let you **bundle logic** with the data it operates on.
* This keeps your code **clean, modular, and easier to understand**.

**✅ 2. Use with interfaces**

* Methods are how types **satisfy interfaces**.
* Without methods, interfaces wouldn’t work.

go

CopyEdit

type Speaker interface {

Speak() string

}

type Dog struct{}

func (d Dog) Speak() string {

return "Woof!"

}

Here, Dog implements the Speaker interface because it has the Speak() method.

**✅ 3. Encapsulation**

* Methods help hide internal logic.
* You can expose behavior without exposing all the internal details of a struct.

**🚀 Quick Recap**

| **Feature** | **Function** | **Method** |
| --- | --- | --- |
| Tied to | Nothing | A specific type (via receiver) |
| Syntax | func greet() | func (p Person) Greet() |
| Use case | General logic | Type-specific behavior |
| Needed for | — | Interfaces, object-like behavior |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Mutex***  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Mutex:***

A **mutex** (short for *mutual exclusion*) is used to **prevent multiple goroutines from accessing shared resources (like variables, maps, or slices) at the same time**. It's part of the sync package and is mainly used to avoid **race conditions**.

**✅ Why use a mutex?**

When multiple goroutines try to read and write shared data **simultaneously**, it can cause **unpredictable behavior**. A mutex helps by **locking** the data when one goroutine is using it, so others have to **wait** until the lock is released.

***Race Condition:***

A **race condition** happens when **two or more goroutines (or threads)** access shared data **at the same time**, and **the final outcome depends on the order** in which they run — which is **unpredictable**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Empty Interface*** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

It **can hold values of any type** — because **all types implement the empty interface**.

**✅ Use Case:**

Use interface{} when you **don’t know the type ahead of time**, or you want to **write generic-like code**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ***Select Statement \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\****

In Go, the select statement is used to wait on multiple channels and choose one that is ready for communication. It is like a switch statement but for channels, allowing you to handle multiple channels concurrently.

Here’s the basic syntax of a select statement:

go

CopyEdit

select {

case channel1 <- value1:

// handle channel1 communication

case channel2 <- value2:

// handle channel2 communication

case value3 := <-channel3:

// handle value received from channel3

default:

// handle case where no channels are ready

}

**Key points about select:**

1. **Non-blocking**: If multiple channels are ready, one of them will be chosen at random.
2. **Blocking**: If none of the channels are ready, the select statement will block (wait) until one of the channels becomes ready.
3. **default case**: If you don’t want to block when none of the channels are ready, you can include a default case, which will execute if no channel is ready.

***Example with multiple channels:***

package main

import (

"fmt"

"time"

)

func main() {

ch1 := make(chan string)

ch2 := make(chan string)

go func() {

time.Sleep(2 \* time.Second)

ch1 <- "message from ch1"

}()

go func() {

time.Sleep(1 \* time.Second)

ch2 <- "message from ch2"

}()

// Use select to listen to both channels

for i := 0; i < 2; i++ {

select {

case msg1 := <-ch1:

fmt.Println("Received:", msg1)

case msg2 := <-ch2:

fmt.Println("Received:", msg2)

}

}

}

**Explanation:**

* We have two channels, ch1 and ch2.
* Two goroutines are used to send messages to these channels after a small delay.
* The select statement listens on both channels and will print the message from whichever channel is ready first.

**Output:**

Received: message from ch2

Received: message from ch1

In this example, ch2 sends its message first because the goroutine for ch2 sleeps for 1 second, while ch1 waits for 2 seconds.

**When to use select:**

* When you're dealing with multiple channels in concurrent programming.
* For handling multiple operations concurrently in an efficient and clean way.
* To avoid blocking on one channel indefinitely if others are ready.