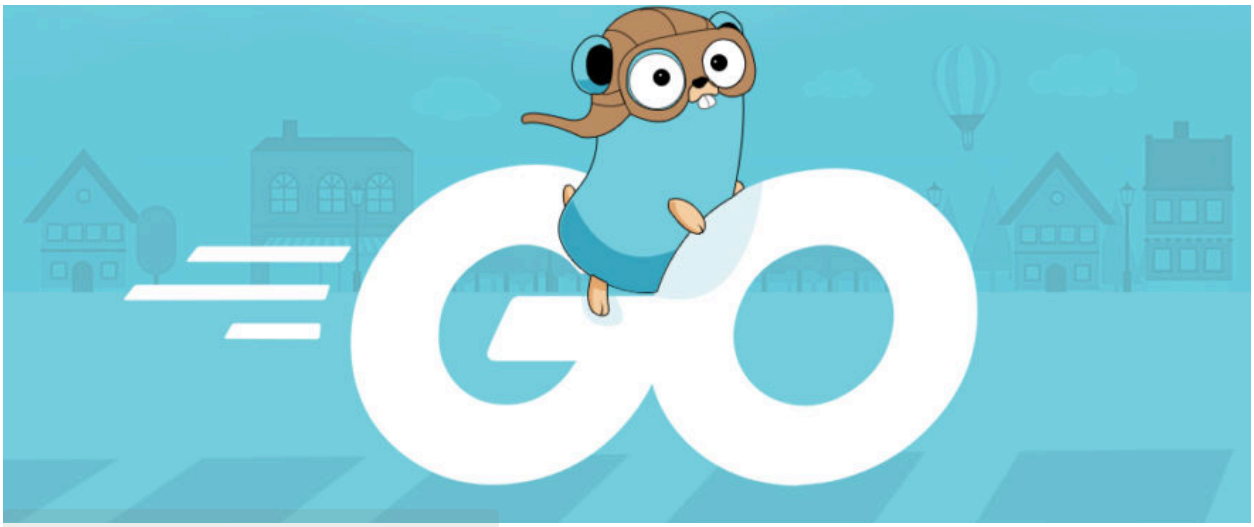

Learning Go

Repository: https://github.com/K6EDWIN/AI-Learning-Go-programming

Directory

```
AI-Learning-Go-programming/
├── bin/            >Compiled executables
├── cmd/
│   └── app.go    >The Site Checker entry point
├── documentation/       >Research logs & Notion backups
├── package/             >Reusable internal logic
├── go.mod               > Module identity
└── README.md
```

1.Project Overview

Project Name: Concurrent Web Scraper Engine

Project Goal: To engineer a high concurrency service monitoring engine that utilizes Go's CSP (Communicating Sequential Processes) model to replace traditional sequential polling methods.

Why Go? I chose Go for its first-class support for concurrency, non-blocking I/O, and its utility first approach to building scalable informatics projects.

Current Sprint: Completion & Deployment

Status: 🟢 Finished

Outcome: Successfully engineered a high-performance concurrent service monitor that replaces traditional sequential polling with the Go CSP model.

2.Quick Summary of the Technology

1. **What is it?** Go is a statically typed, compiled programming language designed at Google for simplicity and high performance concurrency.
2. **Where is it used?** It is a primary language for cloud infrastructure, microservices, and high-performance networking tools.
3. **Real world Example:** Powering major infrastructure like Docker, Kubernetes, and high-speed service monitors.

3.System Requirements

1. **OS:** Linux, Mac, or Windows.
2. **Language Version:** Go 1.25.0 (or 1.23+).
3. **Tools:** VS Code or JetBrains GoLand, Git.

4.Installation & Setup Instructions

1. **Initialize the Module:** `go mod init learning-go`
2. **Verify Environment:** Ensure your `go.mod` file is created to manage the 2026 Go ecosystem.
3. **Run the Project:** `go run cmd/app.go`

5. Minimal Working Example (cmd/app.go)

This example demonstrates a non-blocking concurrent status checker that utilizes Go's CSP model to eliminate I/O wait times.

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

func check(link string, c chan string) {
// 5-second limit so the program doesn't hang
    client := http.Client{Timeout: 5 * time.Second}
    resp, err := client.Get(link)

    // Observation: Checking if err != nil then send a messag
e to the channel and stop
    if err != nil {
        c <- "🔴 " + link + " might be down"
        return
    }

    // close connection
    defer resp.Body.Close()
    c <- "✅ " + link + " up and running"
}

func main() {
    start := time.Now()
    links := []string{
        "https://google.com",
        "https://golang.org",
        "https://github.com",
    }

    // channel to communicate between the main program and Go
routines
```

```
    c := make(chan string)

    for _, link := range links {
        // Launching Goroutines to scale network requests
        go check(link, c)
    }

    for i := 0; i < len(links); i++ {
        fmt.Println(<-c)
    }

    // Calculate and print the total time taken
    fmt.Printf("\nDone🙂 Total time: %v\n", time.Since(star
t))
}
```

6.Strategic Pillars

Pillar	Focus	Relevance
Concurrency	Goroutines, Channels, & sync package.	Eliminating I/O wait times to scale network requests.
Defensive Design	Error handling and Type safety.	Preventing memory leaks and ensuring security.
Performance	Memory allocation and Pointer optimization.	Maximizing throughput with minimal memory footprint.

7.Journal

Date	Module	Concept	Prompt Used	AI Response and Reflection
2026-02-11	Environment	`go mod init`	How do I initialize the module system for a Go project in 2026?	AI recommended `go mod init`. This is fundamental for modern

				dependency management.
2026-02-15	The Communication Pipeline	`http.Get` + Channels	Explain how to use http.Get with Go channels to build a non-blocking checker.	The AI explained that channels are the pipes for communication, allowing for non-blocking requests.
2026-02-18	Synchronization	`WaitGroup`	How do I ensure the main function waits for all concurrent scrapers to finish?	AI introduced `WaitGroup`. It was highly helpful for synchronizing multiple concurrent scrapers effectively.

8.Common Issues & Fixes

1. **Issue: Memory Leaks from Open Connections**
    a. **Fix:** Always use `defer resp.Body.Close()` immediately after checking the error of an HTTP call to ensure resources are freed.
2. **Issue: Hanging Network Requests**
    a. **Fix:** I replaced default `http.Get` with a custom `http.Client` that includes a 5-second timeout to audit and handle failures at the source.
3. **Issue: Synchronization Errors**
    a. **Fix:** I learned to use `WaitGroup` and Channels to prevent the system from crashing or exiting before all concurrent tasks finished.

9. Observations

1. The Verbosity is a Guardrail: In Go, checking `if err != nil` isn't just a chore; it's a security audit. It forces you to handle failure at the source rather than letting it crash the system later.

2. Composition over Inheritance: Since Go has no classes, I’m using Structs for data and Interfaces for behaviour.
3. The Feedback Loop: The compilation speed is a game-changer. It allows for an Iterative Design flow that feels closer to scripting but with the power of a compiled language.