The Problem This Program Solves

Imagine you are a security guard for a building with 5 doors. You need to check if every door is locked.

The "Old" Way (Sequential):

1. You walk to Door 1, check it, and write down the result.
2. Then you walk to Door 2, check it, and write down the result.
3. You repeat this for all 5 doors.

- *Problem:* This is slow. If Door 1 is stuck, you are delayed from checking the other doors.

The "Go" Way (Concurrent - This Program):

1. You sit at a desk (The `main` function).
2. You hire 5 runners (The `Goroutines`).
3. You yell "GO!" and all 5 runners sprint to a different door **at the exact same time**.
4. As soon as a runner checks their door, they shout the result back to you immediately (The `Channel`).
5. You write down the results as they come in.

What the Code Actually Does

1. The Setup (The Checklist)

- The program starts with a list of websites (Google, Clyso, GitHub, etc.). This is your "Doors to check" list.

2. The "Worker" Definition (`func checkUrl`)

- This is the instruction manual for the runners. It tells them:
  - "Start a stopwatch."
  - "Go to this website."
  - "If it works, write down 'UP' and stop the watch."
  - "If it fails, write down 'DOWN'."
  - "Send the report back to the main desk."

3. The Launch (`go checkUrl`)

- This is the magic part. The program loops through your list of websites.
- Instead of visiting the first site and waiting, it uses the command `go`. This spawns a "background worker" to go visit that site while the main program immediately spawns the next worker.
- Within milliseconds, 5 separate workers are checking 5 separate sites simultaneously.

4. The Collection (`<-c`)

- The main program pauses and waits for messages to come back.
- Because the checks happened at the same time, the results might come back out of order (the fastest website reports back first, not necessarily the first one in the list).

The imported things:

1. `fmt` (The "Printer")

- **Full Name:** Format
- **What it does:** It handles input and output (text). It allows **our** program to "speak" to **us** through the terminal.

2. `net/http` (The "Web Browser")

- **Full Name:** Network / Hypertext Transfer Protocol
- **What it does:** This is the tool that lets **our** Go program connect to the internet. It can act like a web server or a web client (browser).
- **In our code:**
  - **We** use `http.Get(url)` to actually visit the website.

3. `time` (The "Stopwatch")

- **Full Name:** Time
- **What it does:** It measures time, handles dates, and deals with durations.
- **In our code:**
  - **We** use `start := time.Now()` to start a stopwatch right before **we** check a site.
  - **We** use `time.Since(start)` to stop the watch and calculate exactly how many milliseconds the website took to respond (Latency).

The channel

c := make(chan Result)

1. **c**: This is just the name **we** gave the channel
2. **make**: This is the command to build something new in memory. **We** are telling the computer: "Construct this object for **us**."
3. **chan**: This tells Go **what** kind of thing to build. **We** are building a **Channel** (a communication pipe).
4. **Result**: This is the most important part. It tells the pipe **what fits inside**.
   - **We** defined a Result earlier (the box holding the URL, Status, and Latency).
   - This channel is **strictly** for Result objects.

The Routines

In Go, when **we** say "routines," **we** are talking about **Goroutines**.

Think of a "Routine" (Goroutine) as an **independent worker**.

The "Main" Routine (Us)

- When **we** start the program, **we** are the **Main Routine**.
- **We** are the boss. **We** read the code line-by-line.

The "Go" Routines (The Workers)

- When **we** use the command `go checkUrl(...)`, **we** are hiring a new worker.
- **We** tell this worker: *"Take this URL and go check it. Don't wait for me."*
- This worker is a **Goroutine**.

The Concurrency

```
// Spin up a Goroutine for each URL
    for _, url := range urls {
        go checkUrl(url, c)
    }
```

This is the engine room of **our** program. This tiny block of code is what makes **us** "Concurrent" instead of just "Sequential."

Let's break it down into three pieces: The Loop, The "Trash Can," and The Launch.

1. The Loop (`for ... range`)

In Go, `range` is a keyword that walks through a list one item at a time.

- **We** are telling the computer: *"Take our list of `urls` and give **us** each item, one by one."*

2. The "Trash Can" (_)

This is a quirky Go rule. When **we** use `range`, it actually gives **us** *two* things back for every item:

1. **The Index:** The position number (0, 1, 2, 3...).
2. **The Value:** The actual data (e.g., "https://www.google.com/search?q=google.com").

We only care about the URL. **We** don't care if it is number 0 or number 4.

- **Problem:** Go is strict. If **we** create a variable for the index (like `i`) and don't use it, the program will crash with an error: *"Variable declared and not used."*
- **Solution:** The Underscore _. This is **our** "Trash Can."
    - **We** are saying: *"Put the index number in the trash (_), and put the website address into the variable `url`."*

3. The Launch Command (`go`)

This is the most powerful keyword in the entire language.

- **Without `go`:**

  ```
  checkUrl(url, c)
  ```

  If **we** wrote this, **we** (the Main program) would stop. **We** would walk to the website, check it, come back, and *then* loop to the next one. This is slow.

- **With `go`:**

  ```
  go checkUrl(url, c)
  ```

> **We** are saying: *"Create a **new** background worker (a Goroutine). Give them this `url` and this channel `c`. Let them run on their own. **We** are moving immediately to the next line."*

The Result: Because **we** used `go`, this loop finishes in microseconds. **We** have successfully spawned 5 separate workers who are all running at the same time, while **we** move on to the next part of the code.

```
result := <-c
```

This line says:

"Wait here until a message comes out of channel `c`. When it does, grab it, create a new variable named `result`, and store the message inside it."

The Critical Concept: "Blocking" (The Wait)

This is the most important part to understand.

When **we** run this line, if the channel is empty (because the websites are still loading), **we stop**.

- **We** do not move to the next line.
- **We** do not pass Go.
- **We** freeze right there.

We remain frozen until **one** of the background workers finishes and pushes a value into the pipe. The moment that happens:

1. **We** wake up.
2. **We** grab that value.
3. **We** put it in `result`.
4. **We** move to the next line (printing it).

Why is this cool?

Because **we** don't need to constantly ask "Are you done yet? Are you done yet?" **We** just sit back and relax. The code automatically wakes **us** up exactly when there is work to do.

THE DOCKERFILE

1. FROM golang:alpine

- **The Translation:** *"Start with a computer that already has Go and Alpine Linux installed."*
- **What it does:** Instead of installing Windows, then downloading Chrome, then downloading Go... **we** are downloading a pre-made system image that already has everything **we** need.
- **Why "Alpine"?** Alpine is a tiny, lightweight version of Linux (like 5MB). It makes **our** container download very fast.

2. WORKDIR /app

- **The Translation:** *"Create a folder named /app and go inside it."*
- **What it does:** It sets up **our** workspace. Any command **we** run after this line will happen inside this specific folder. It keeps **our** container organized so **we** don't leave files checking in the root directory.

3. COPY . .

- **The Translation:** *"Copy everything from HERE to THERE."*
- **The First Dot (.):** This means **our** current folder on **our** actual computer (where main.go lives).
- **The Second Dot (.):** This means the current folder inside the container (which is /app because of the previous line).
- **The Result:** Docker takes a snapshot of **our** code and puts it inside the container.

4. RUN go build -o monitor main.go

- **The Translation:** *"Turn our text code into a working program."*
- **What it does:** This runs the Go compiler *inside* the container.
  - go build: The command to build.
  - -o monitor: "Name the output file 'monitor'."
  - main.go: The file to build.
- **The Result: We** now have a runnable file named monitor inside the /app folder.

5. CMD ["./monitor"]

- **The Translation:** *"When the container starts, push this button."*
- **The Difference:**
  - RUN happens **once** when **we** build the image (baking the cake).
  - CMD happens **every time we** run the container (eating the cake).
- **What it does:** It tells Docker: "As soon as this container wakes up, run the ./monitor program immediately."