Phase 2

Intermediate Go Concepts

Packages and Modules

- Standard Library: https://pkg.go.dev/std
- Creating your own package
- Using third-party packages: https://pkg.go.dev/

Error Handling

error Type

Go uses a built-in error interface to handle errors explicitly.

```
func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, errors.New("cannot divide by zero")
    }
    return a / b, nil
}
```

Custom error types

```
import "fmt"

type MathError struct {
    Operation string
    Reason    string
}

func (e MathError) Error() string {
    return fmt.Sprintf("Error in %s: %s", e.Operation, e.Reason)
}

func divide(a, b float64) (float64, error) {
    if b == 0 {
        return 0, MathError{"divide", "cannot divide by zero"}
    }
    return a / b, nil
}

func main() {
    _, err := divide(5, 0)
    if err != nil {
        fmt.Println(err)
    }
```

panic and recover

```go
package main

import "fmt"

func mayPanic() {
    panic("Something went wrong!")
}

func safeCall() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from panic:", r)
        }
    }()
    mayPanic()
    fmt.Println("This will not be printed if panic is not recovered.")
}

func main() {
    safeCall()
    fmt.Println("Program continues after recovery.")
}
```

- **panic()**

causes a run-time error and exits the function stack.

- **defer + recover()**

allows you to catch the panic and continue.

Interfaces and Polymorphism

Concurrency

```go
package main

import (
    "fmt"
    "time"
)

func sayHello() {
    fmt.Println("Hello from goroutine")
}

func main() {
    go sayHello()           // Run concurrently
    time.Sleep(time.Second) // Wait for goroutine
    fmt.Println("Main ends")
}
```

Use ‘go’ keyword for run concurrently

Channels

- Channels are pipes for communication between goroutines.
- Use chan to declare.
- Channels are typed.

  - ch <- sends data
  - <- ch receives data
  - If there is no go routine using the channel the program will deadlock

```go
func sayHi(ch chan string) {
    ch <- "Hi from goroutine"
}

func main() {
    ch := make(chan string)
    go sayHi(ch)
    msg := <-ch
    fmt.Println(msg)
}
```

Buffered vs Unbuffered Channels

- **Unbuffered:**
  - **Sends block until the receiver is ready.**
- **Buffered:**
  - **Allows limited queue before blocking.**

```go
func main() {
    ch := make(chan string, 2) // Buffered
    ch <- "A"
    ch <- "B"
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Select Statement

- select lets you wait on multiple channel operations.

```go
func main() {
    ch1 := make(chan string)
    ch2 := make(chan string)

    go func() { ch1 <- "one" }()
    go func() { ch2 <- "two" }()

    select {
    case msg1 := <-ch1:
        fmt.Println("Received:", msg1)
    case msg2 := <-ch2:
        fmt.Println("Received:", msg2)
    }
}
```

```go
func serviceA(ch chan string) {
    time.Sleep(2 * time.Second) // simulate delay
    ch <- "Response from Service A"
}

func serviceB(ch chan string) {
    time.Sleep(1 * time.Second) // faster service
    ch <- "Response from Service B"
}

func handler(c *gin.Context) {
    chA := make(chan string)
    chB := make(chan string)

    go serviceA(chA)
    go serviceB(chB)

    select {
    case res := <-chA:
        c.JSON(http.StatusOK, gin.H{"source": "A", "data": res})
    case res := <-chB:
        c.JSON(http.StatusOK, gin.H{"source": "B", "data": res})
    case <-time.After(3 * time.Second):
        c.JSON(http.StatusGatewayTimeout, gin.H{"error": "Timeout waiting for services"})
    }
}

func main() {
    r := gin.Default()
    r.GET("/data", handler)

    fmt.Println("Server running at http://localhost:8080")
    r.Run(":8080")
}
```

- Which channel done first it will go to that case

sync Package: Mutex & WaitGroup

- WaitGroup
  - Waits for a group of goroutines to finish.
- Mutex
  - Provides mutual exclusion (safe access to shared data).

```
var wg sync.WaitGroup

wg.Add(1)
go func() {
    fmt.Println("Goroutine")
    wg.Done()
}()
wg.Wait()
```

```
var mu sync.Mutex
count := 0

mu.Lock()
count++
mu.Unlock()
```

Real use case for waitgroup & mutex

1. Using WaitGroup for Concurrent API Calls (Fan-out Pattern)
   - Assume that we are fetching data from mq

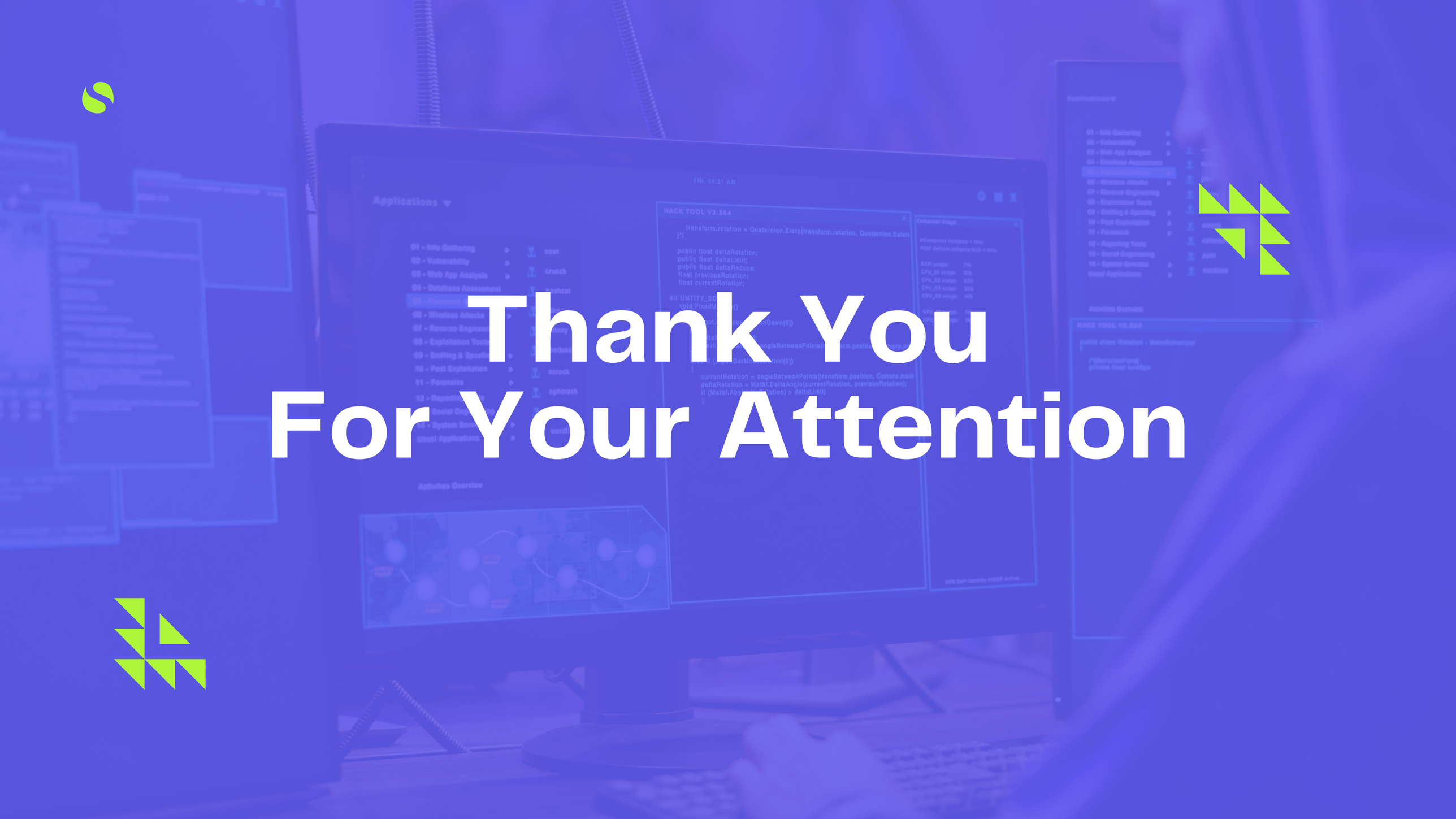
```go
func fetchFromAPI(name string, wg *sync.WaitGroup) {
    defer wg.Done()
    time.Sleep(1 * time.Second)
    fmt.Println("Fetched from", name)
}

func main() {
    var wg sync.WaitGroup
    apis := []string{"User", "Orders", "Notifications"}

    for _, api := range apis {
        wg.Add(1)
        go fetchFromAPI(api, &wg)
    }

    wg.Wait()
    fmt.Println("All APIs fetched")
}
```

Real use case for waitgroup & mutex

2. Using Mutex to Prevent Race Conditions

- Assume that multiple client editing the same data in db

```go
type User struct {
    Name  string
    Email string
}

// Simulate in-memory DB
var (
    db = map[string]User{
        "john": {"John Doe", "john@example.com"},
    }
    mu sync.Mutex
)

func updateEmail(username, newEmail string, wg *sync.WaitGroup) {
    defer wg.Done()

    mu.Lock()
    user, exists := db[username]
    if exists {
        user.Email = newEmail
        db[username] = user
        fmt.Println("Updated:", username)
    } else {
        fmt.Println("User not found:", username)
    }
    mu.Unlock()
}

func main() {
    var wg sync.WaitGroup

    wg.Add(2)
    go updateEmail("john", "new1@example.com", &wg)
    go updateEmail("john", "new2@example.com", &wg)

    wg.Wait()

    fmt.Println("Final record:", db["john"])
}
```


Thank You
For Your Attention