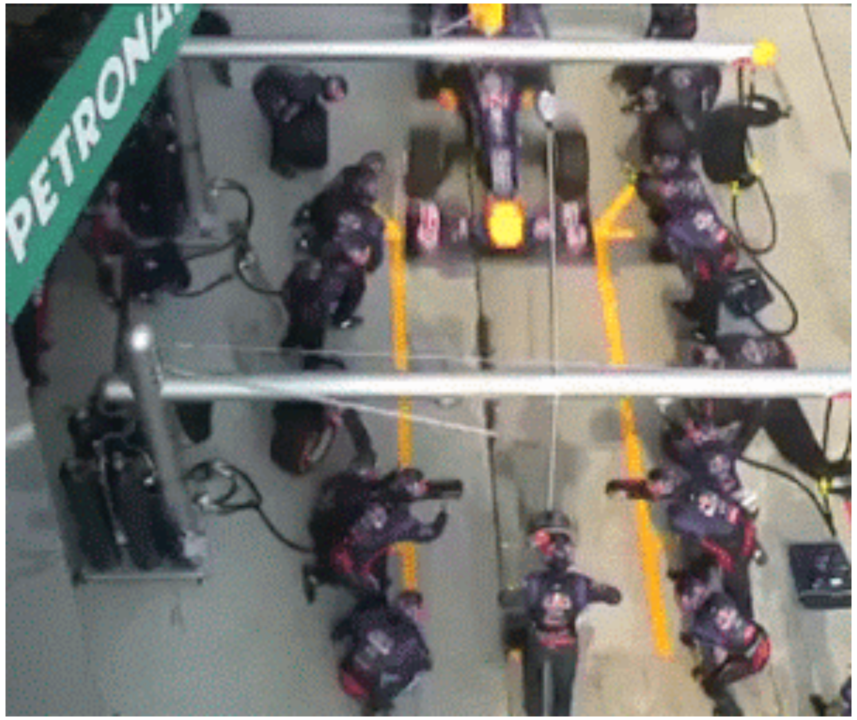
Helsinki Gophers Meetup
06 November 2024

Graceful Shutdown

Bengisu Kandemir
Jr. Backend Developer at SSH Communications Security

Safeguarding Software Integrity

- Isolation of components
- Database transactions and rollbacks
- …
- Graceful shutdown



https://www.reddit.com/r/oddlysatisfying/comments/3aiyhe/pit_stop_efficiency/

Graceful shutdown ensures data integrity

- ✓ Completion of all background processes
- ✓ Completion of requests in process
- ✓ Rejection of new requests

Graceful Shutdown in Go

LISTEN TERMINATION SIGNAL → PROPAGATE THE SIGNAL TO MULTIPLE GOROUTINES → WAIT FOR ALL RUNNING GOROUTINES TO EXIT + SERVER SHUTDOWN

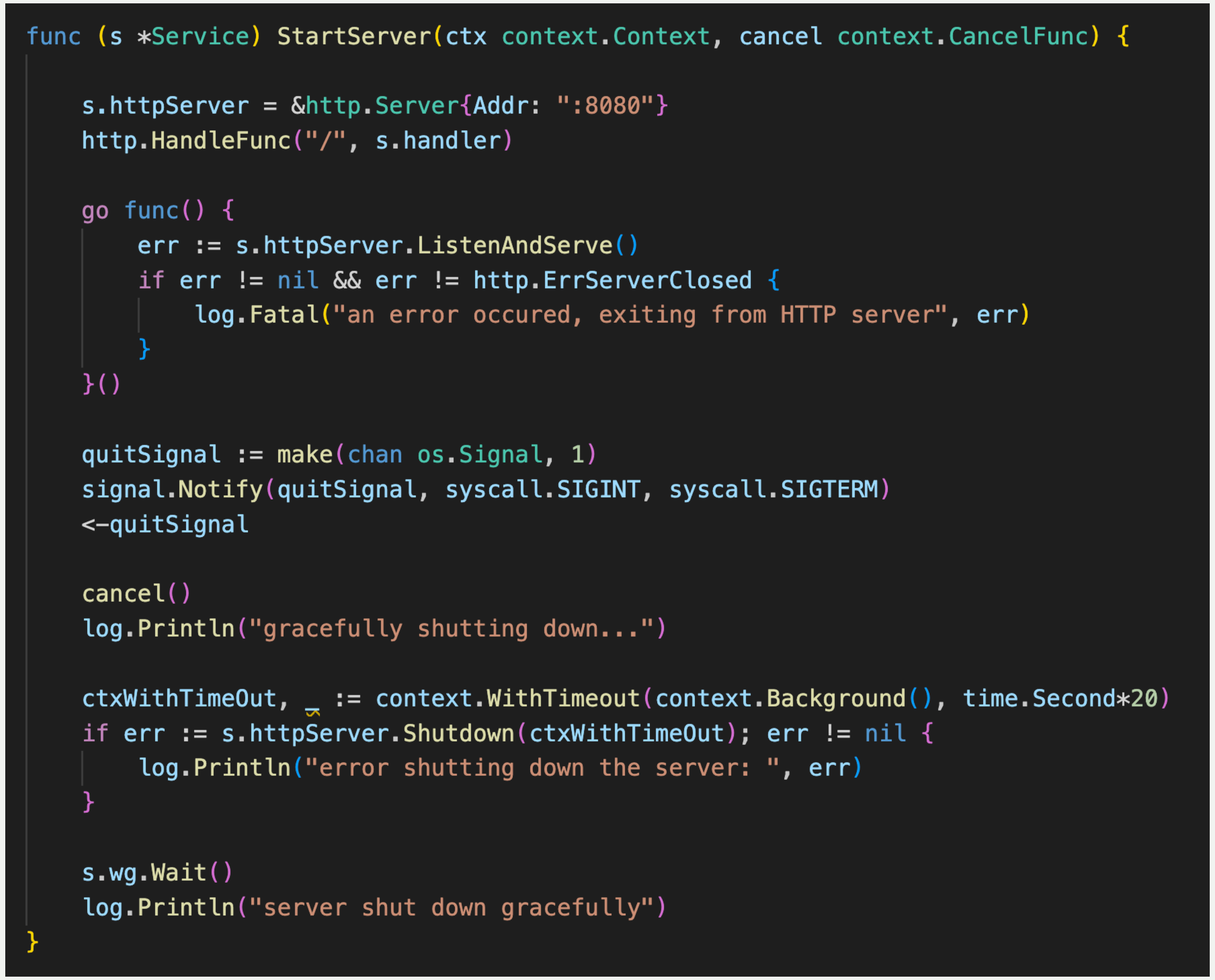
Channels

Context

WaitGroups

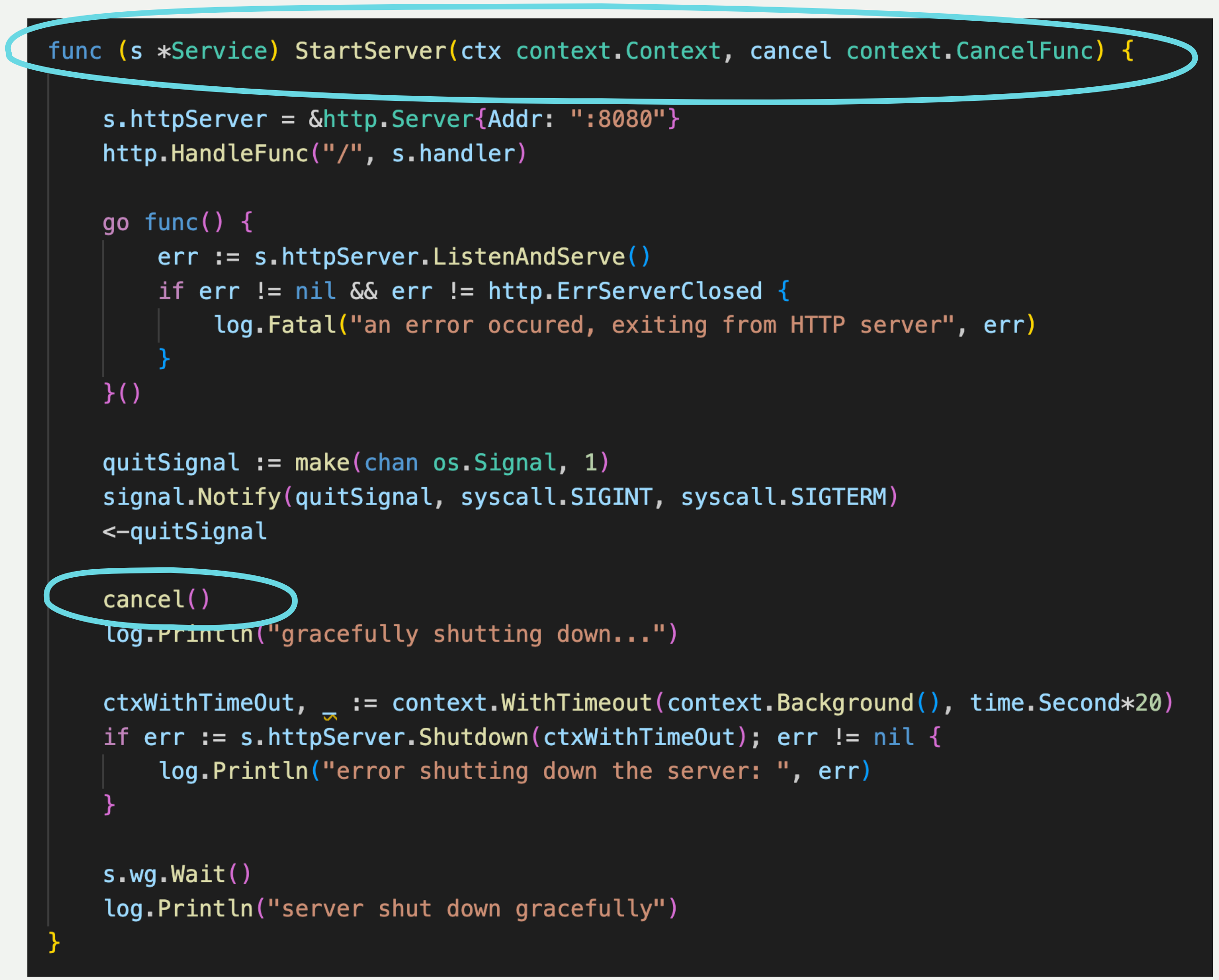
(*Server)Shutdown

Inside the StartServer Function

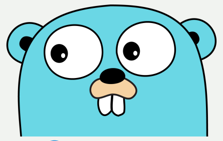


```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

Inside the StartServer Function

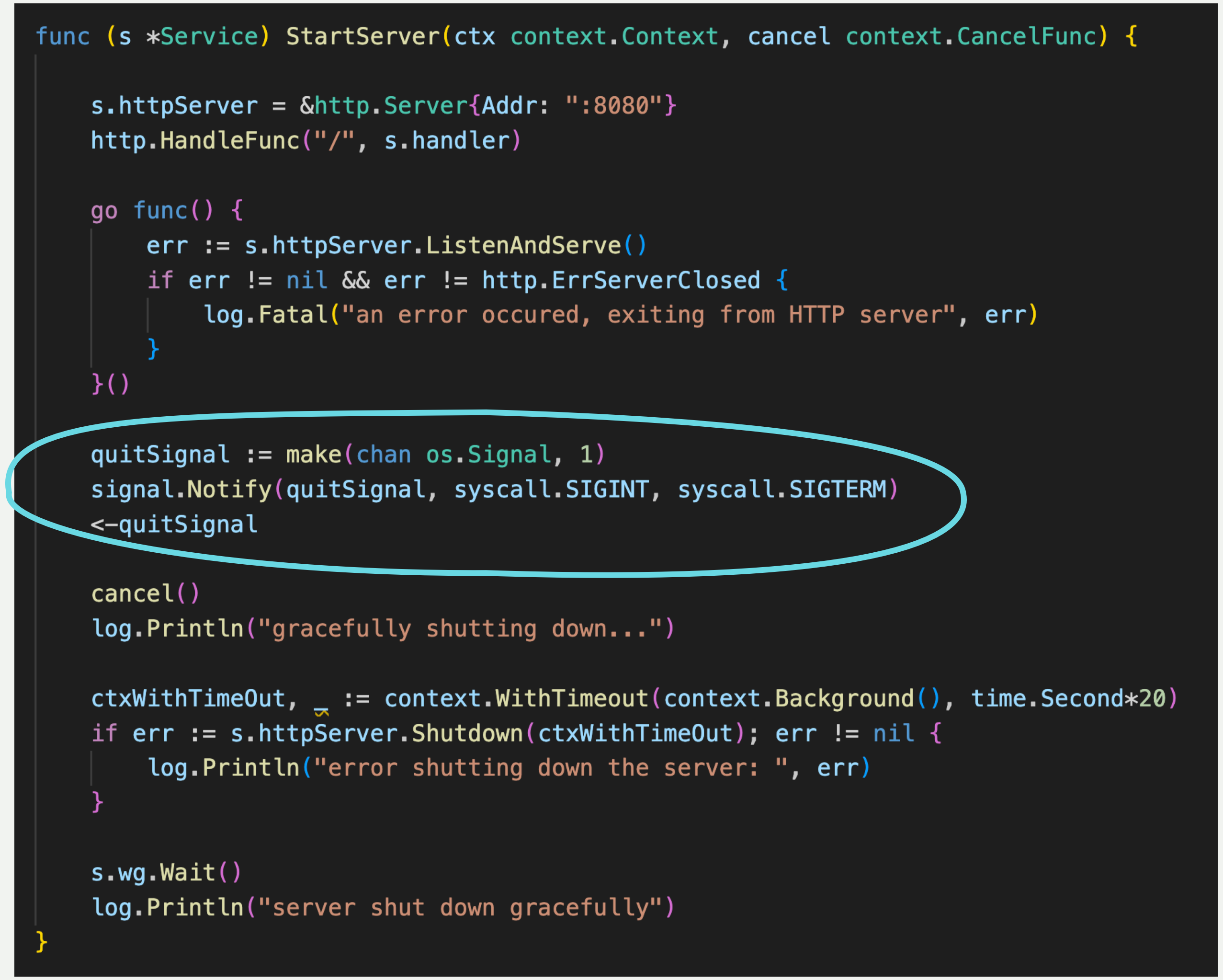


```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

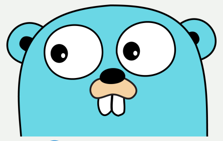


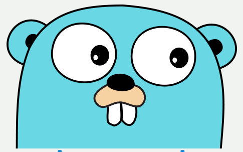
Context

Inside the StartServer Function

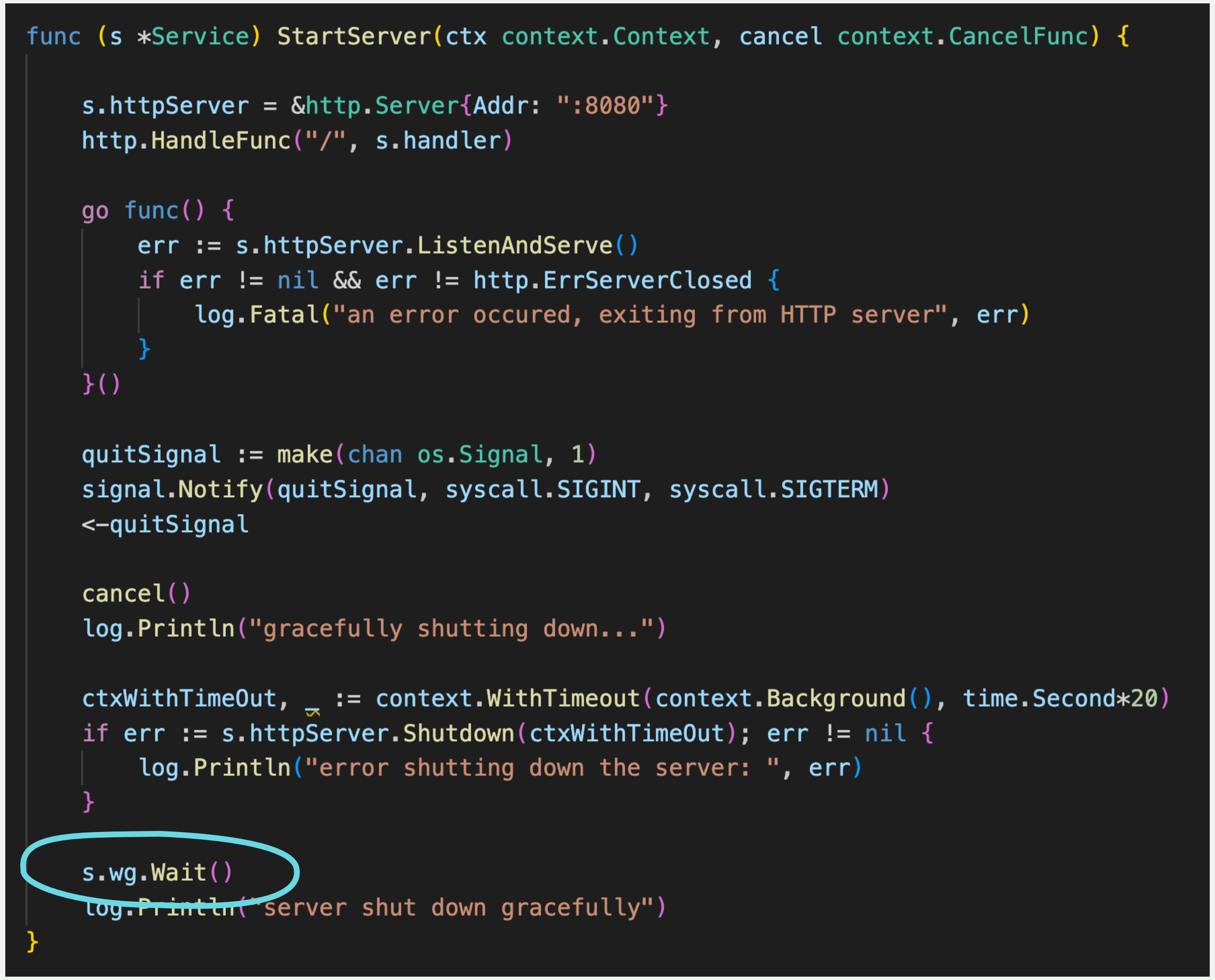


```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

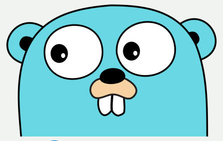


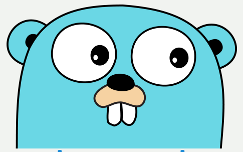


Inside the StartServer Function



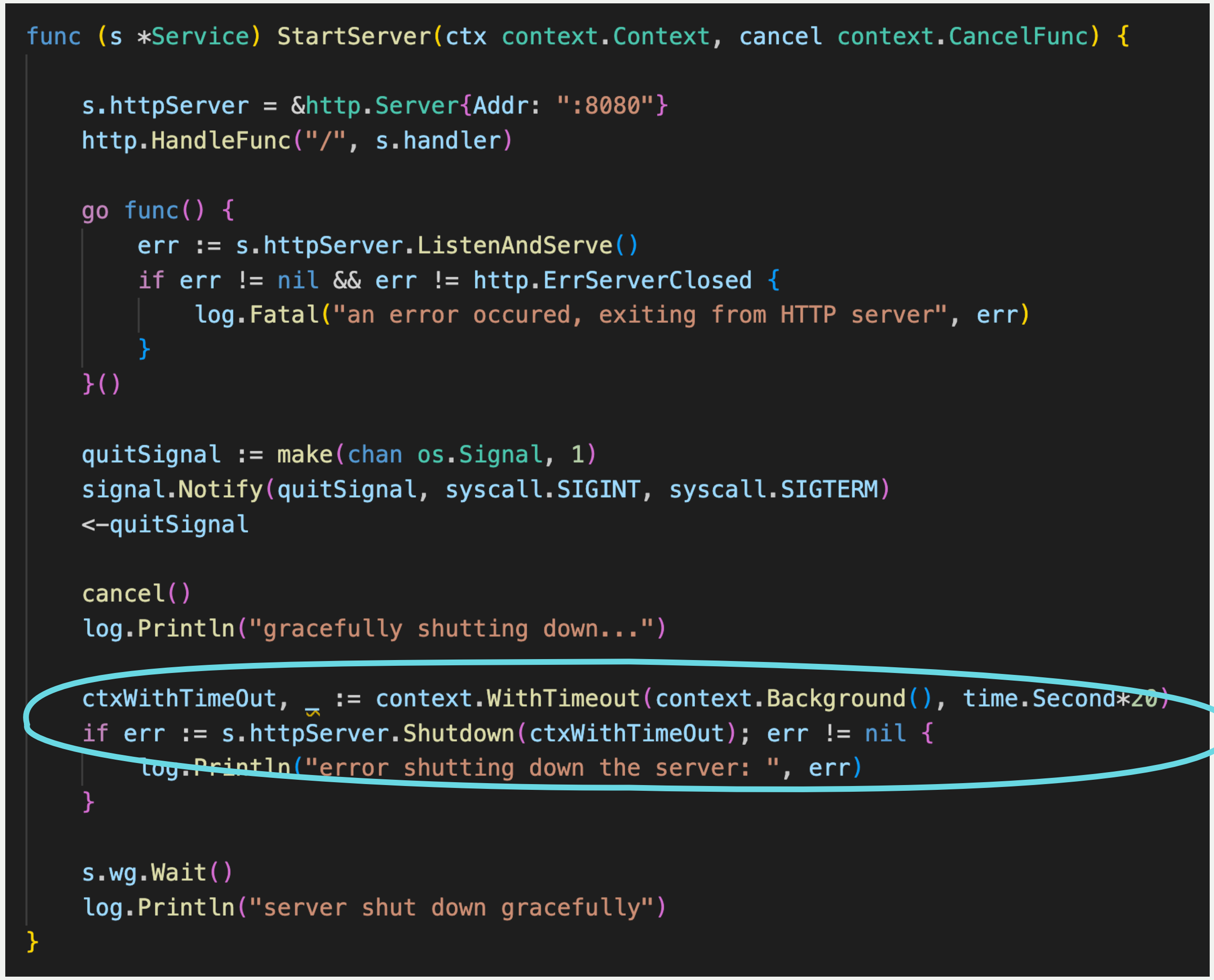
```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```


Context

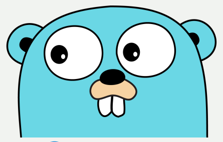

Channels

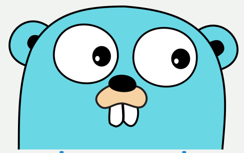
WaitGroups

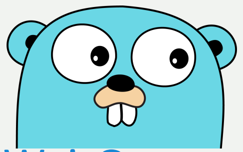
Inside the StartServer Function



```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

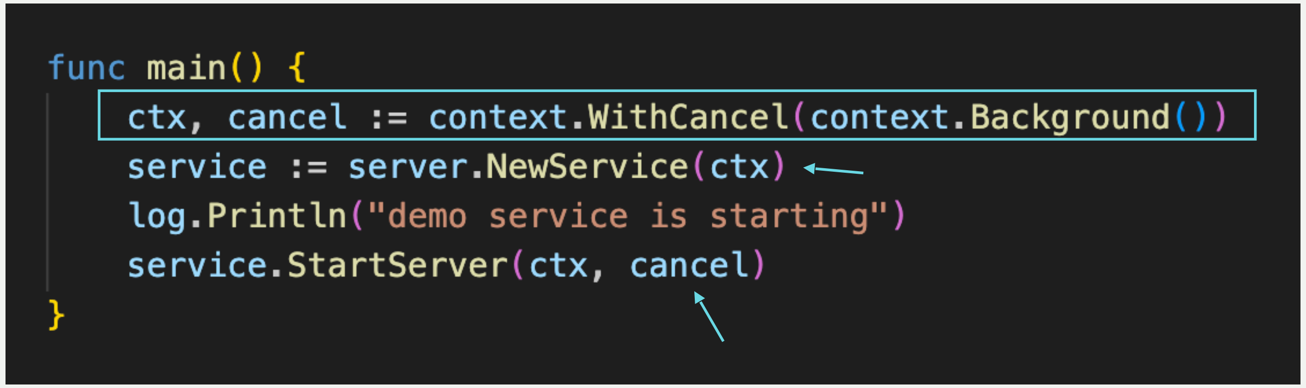




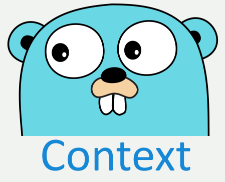


Context in Go passes information

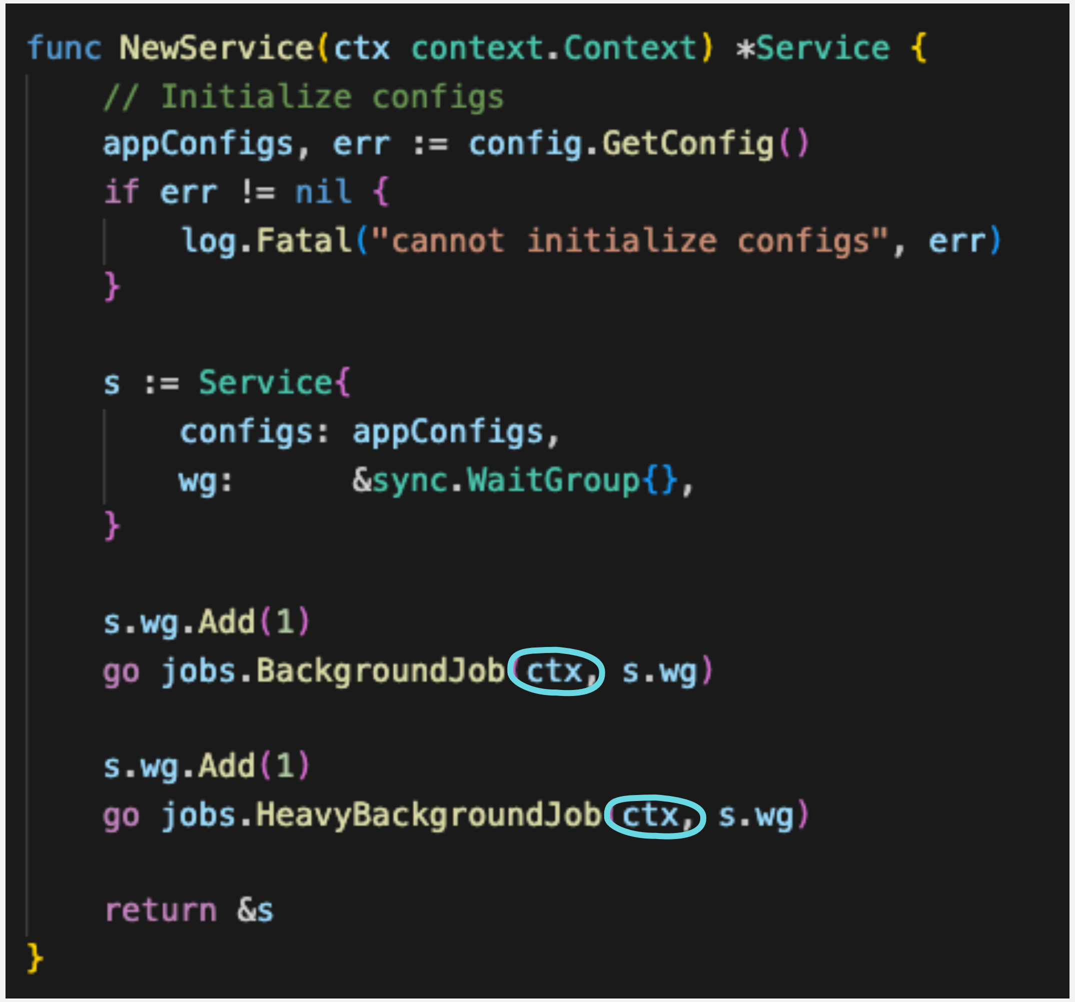
- In the main of the service, we start with creating a context with cancel function:



```go
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    service := server.NewService(ctx)
    log.Println("demo service is starting")
    service.StartServer(ctx, cancel)
}
```

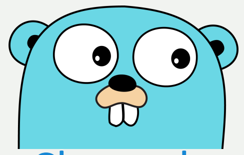


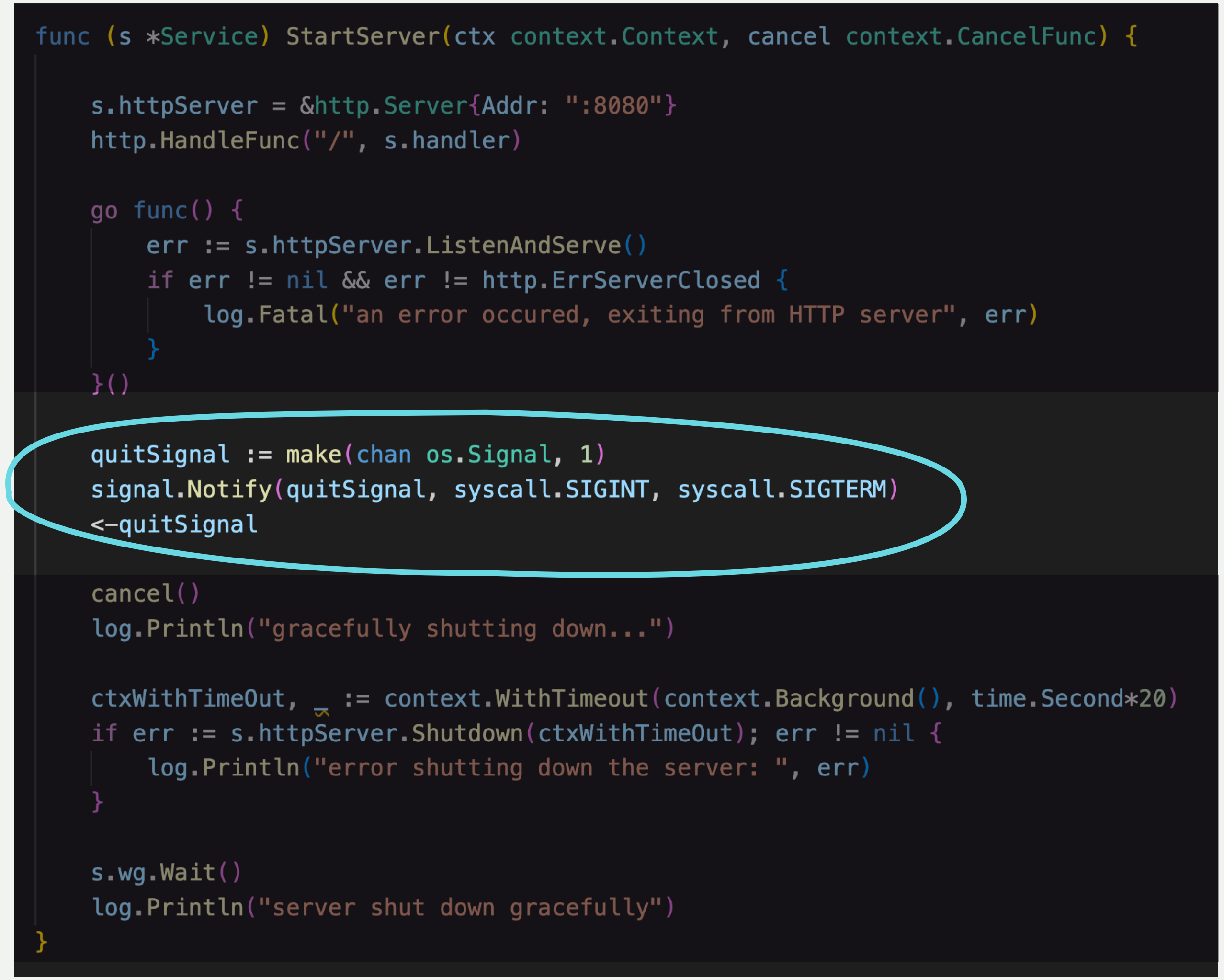
Context in Go passes information



```go
func NewService(ctx context.Context) *Service {
    // Initialize configs
    appConfigs, err := config.GetConfig()
    if err != nil {
        log.Fatal("cannot initialize configs", err)
    }

    s := Service{
        configs: appConfigs,
        wg:      &sync.WaitGroup{},
    }

    s.wg.Add(1)
    go jobs.BackgroundJob(ctx, s.wg)

    s.wg.Add(1)
    go jobs.HeavyBackgroundJob(ctx, s.wg)

    return &s
}
```

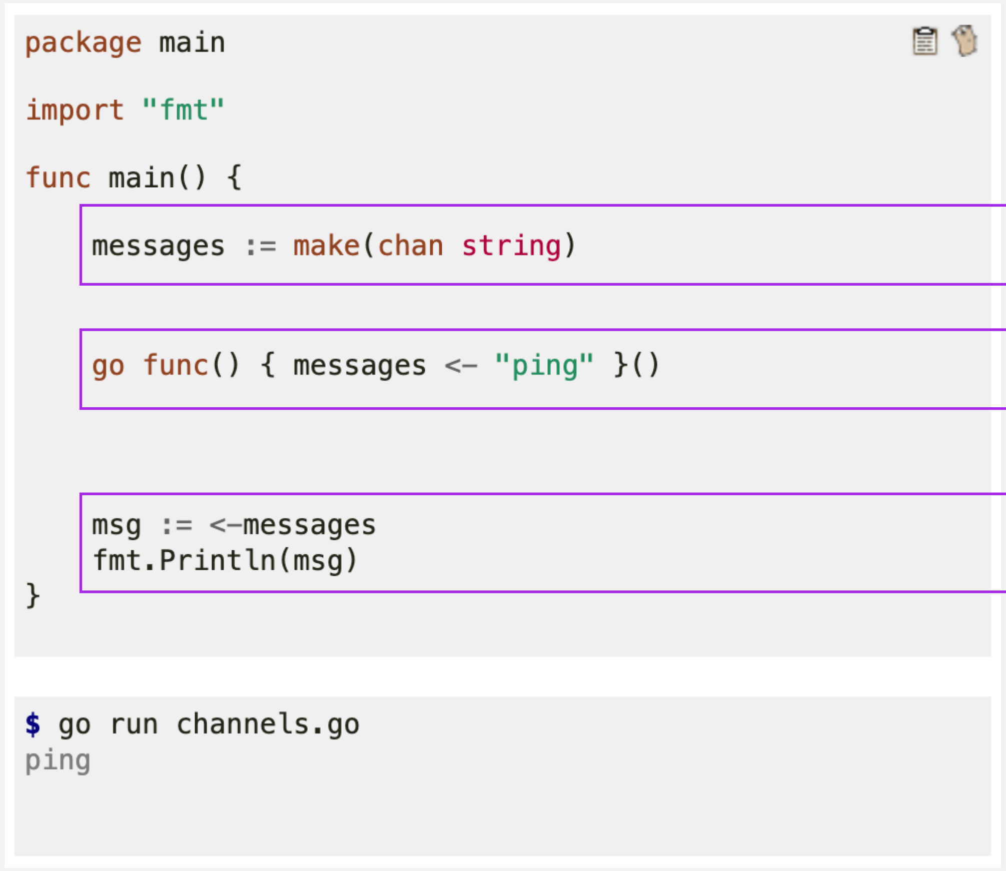
How to listen the termination signal?


Channels



```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

Channels in Go are pipelines



```
package main

import "fmt"

func main() {
    messages := make(chan string)

    go func() { messages <- "ping" }()


    msg := <-messages
    fmt.Println(msg)
}
```

```
$ go run channels.go
ping
```

Create channel

Send value to the channel

Receive value from the channel

https://gobyexample.com/channels

How to listen the termination signal?

Channels

Inside the StartServer function

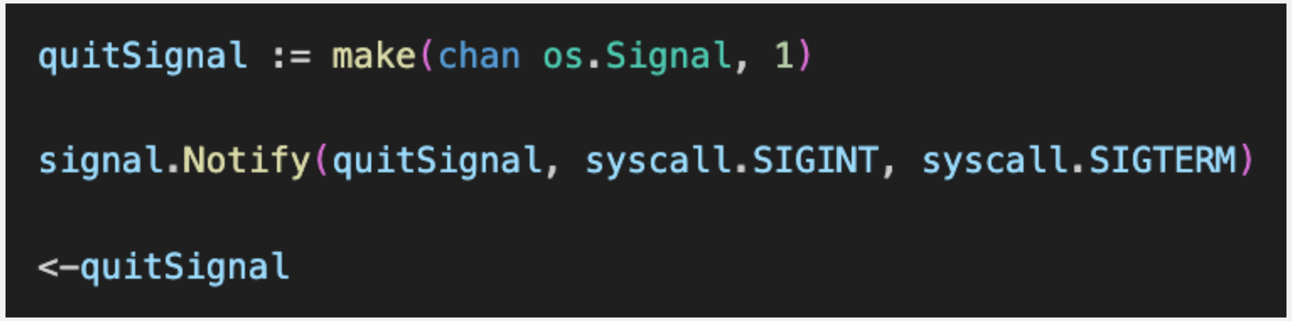
```
quitSignal := make(chan os.Signal, 1)

signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)

<-quitSignal
```

Create channel for signal

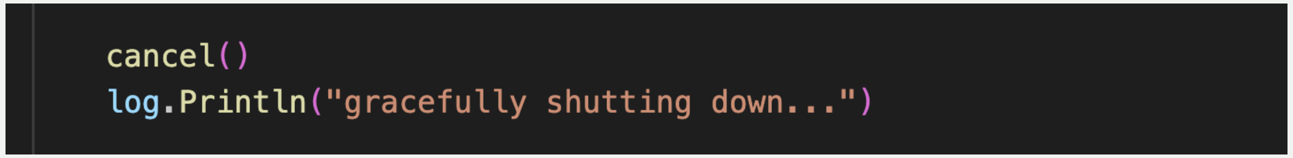
quitSignal <- SIGINT or SIGTERM

Block until receiving the signal

After receiving the termination signal



```go
quitSignal := make(chan os.Signal, 1)

signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)

<-quitSignal
```



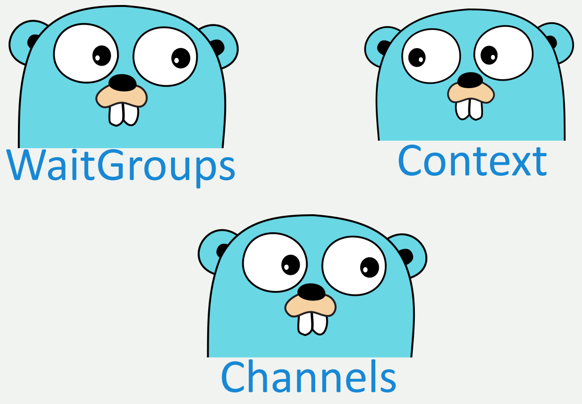
```go
    cancel()
    log.Println("gracefully shutting down...")
```

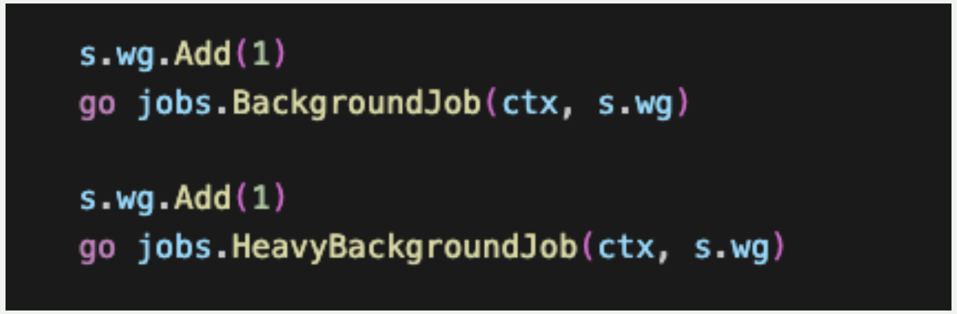
+

WAIT FOR ALL RUNNING GOROUTINES TO EXIT

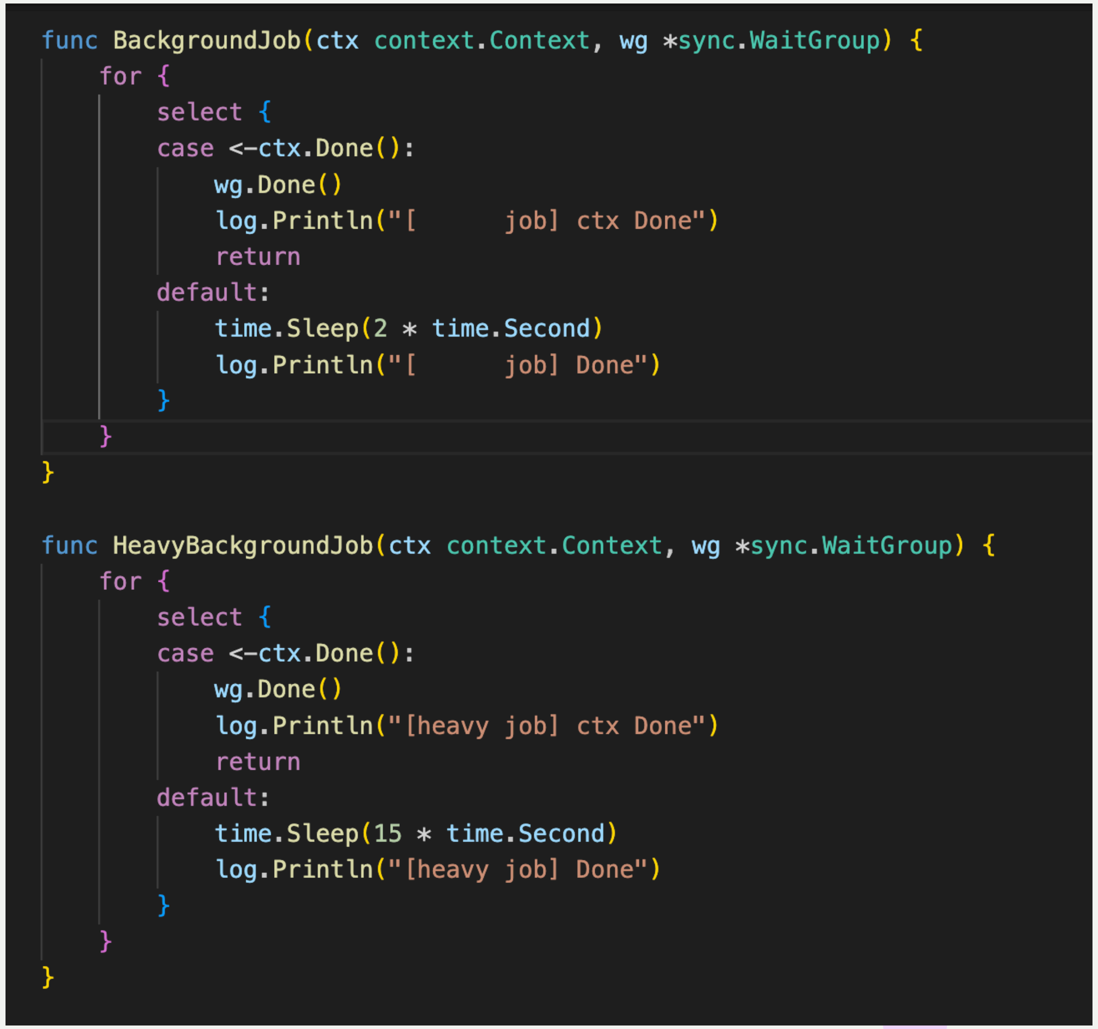
SERVER SHUTDOWN

Graceful shutdown of background processes



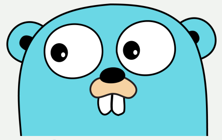


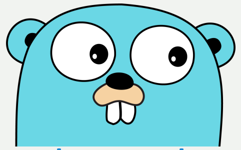
```go
s.wg.Add(1)
go jobs.BackgroundJob(ctx, s.wg)

s.wg.Add(1)
go jobs.HeavyBackgroundJob(ctx, s.wg)
```

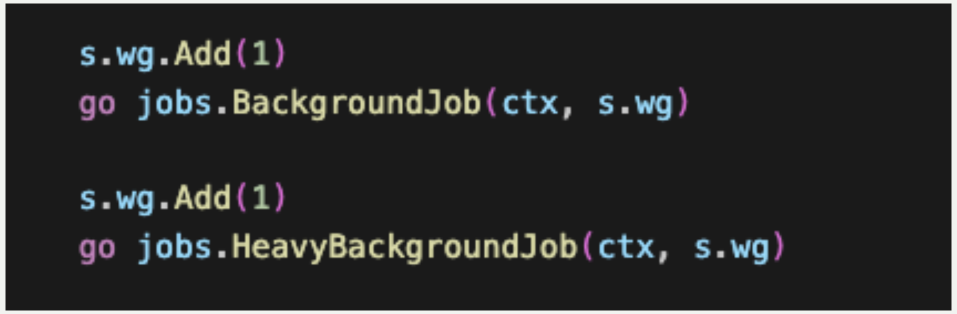


```go
func BackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[      job] ctx Done")
            return
        default:
            time.Sleep(2 * time.Second)
            log.Println("[      job] Done")
        }
    }
}

func HeavyBackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[heavy job] ctx Done")
            return
        default:
            time.Sleep(15 * time.Second)
            log.Println("[heavy job] Done")
        }
    }
}
```

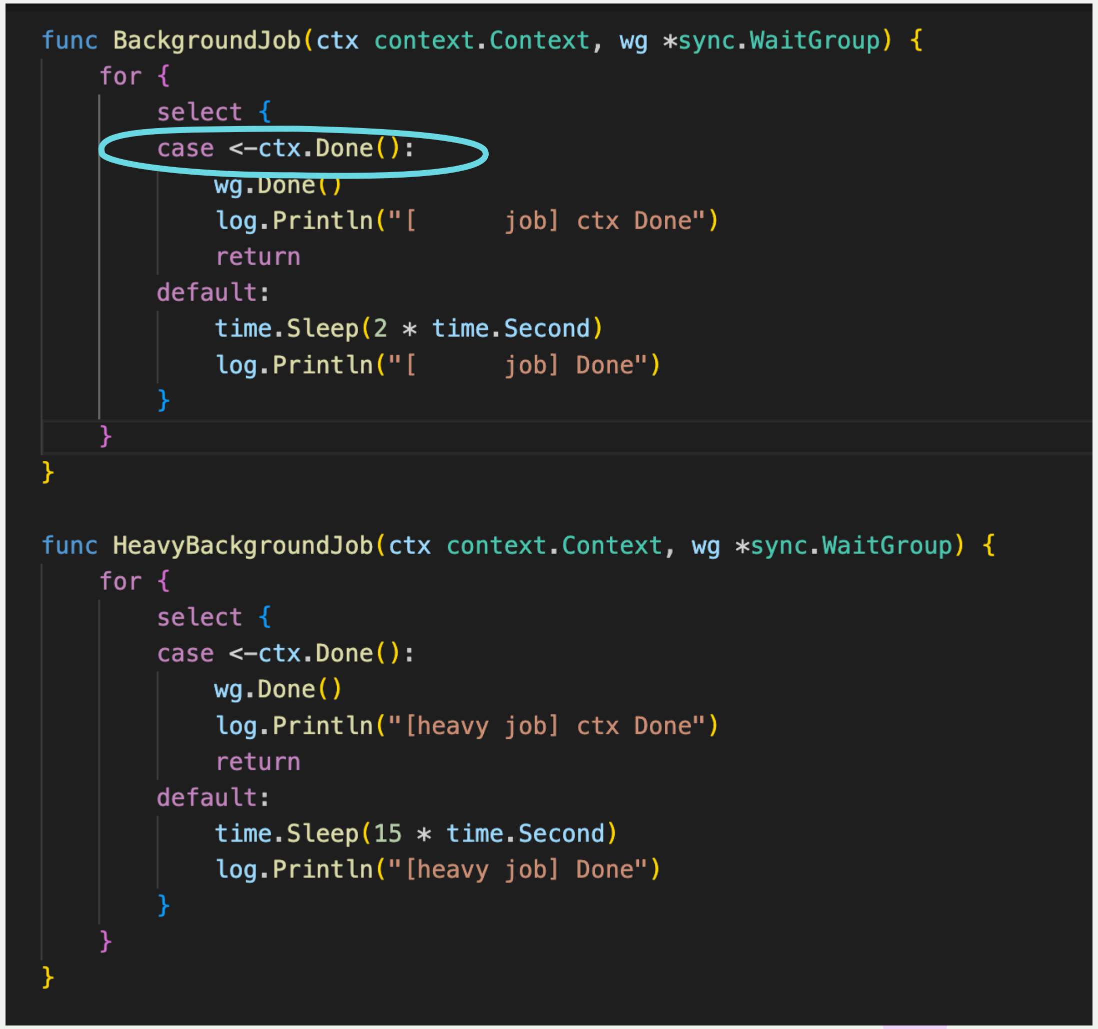
Graceful shutdown of background processes





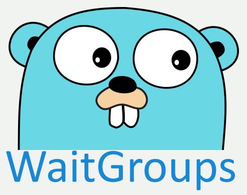


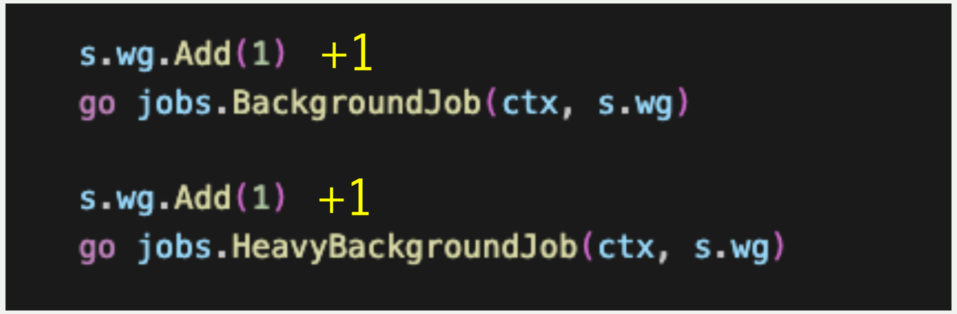
```go
s.wg.Add(1)
go jobs.BackgroundJob(ctx, s.wg)

s.wg.Add(1)
go jobs.HeavyBackgroundJob(ctx, s.wg)
```



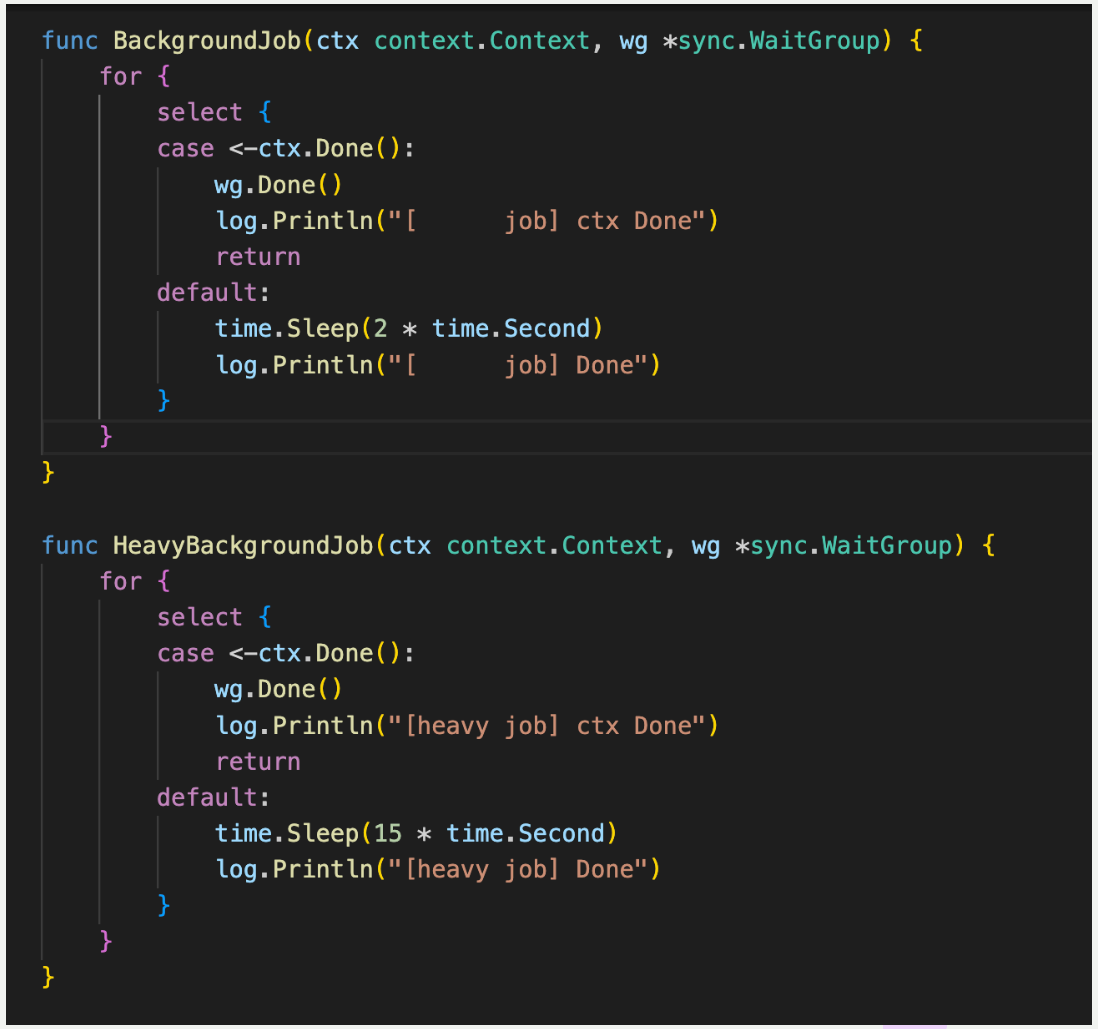
```go
func BackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[      job] ctx Done")
            return
        default:
            time.Sleep(2 * time.Second)
            log.Println("[      job] Done")
        }
    }
}

func HeavyBackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[heavy job] ctx Done")
            return
        default:
            time.Sleep(15 * time.Second)
            log.Println("[heavy job] Done")
        }
    }
}
```

Graceful shutdown of background processes



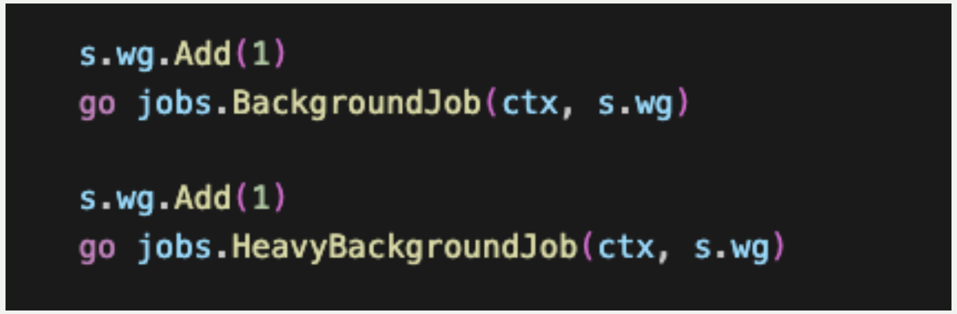


```
s.wg.Add(1)  +1
go jobs.BackgroundJob(ctx, s.wg)

s.wg.Add(1)  +1
go jobs.HeavyBackgroundJob(ctx, s.wg)
```

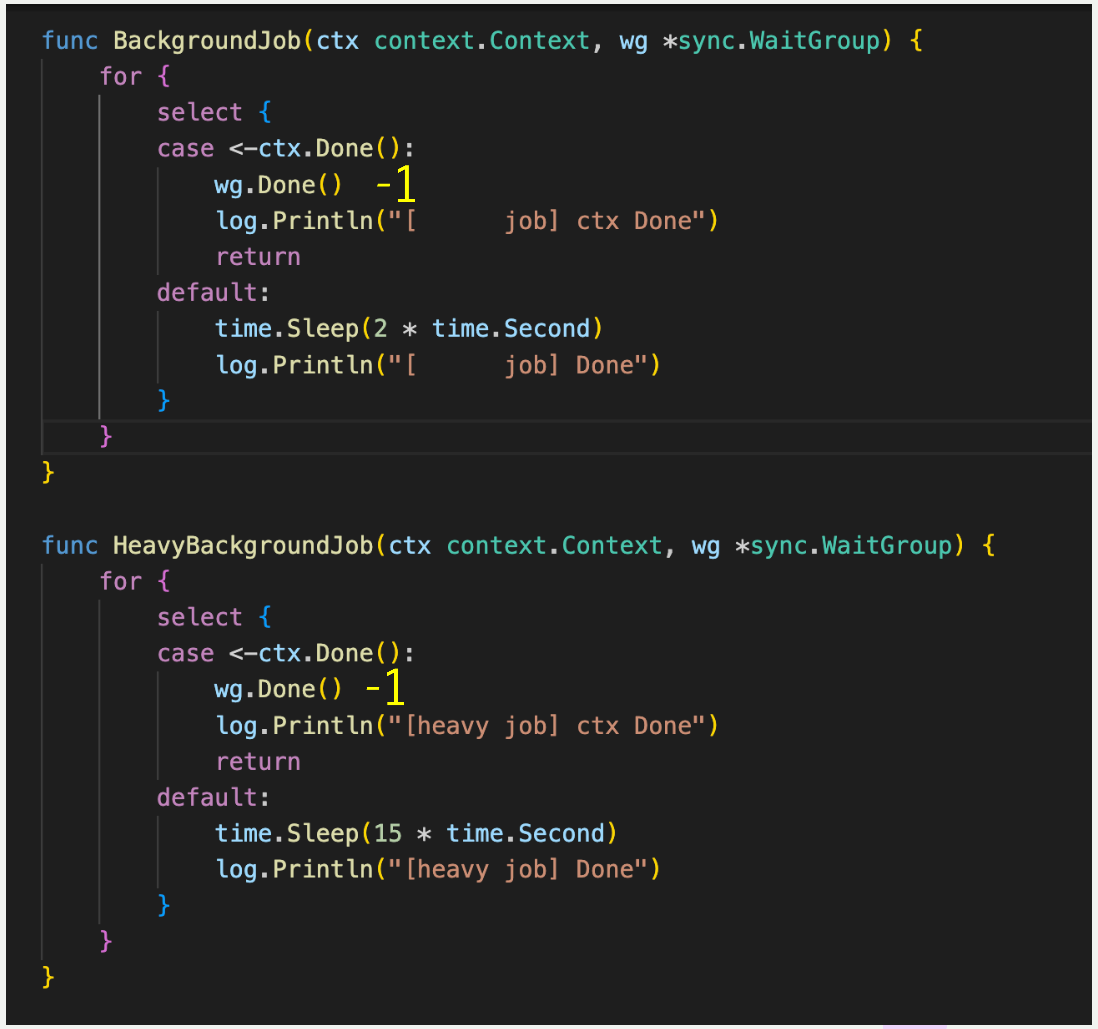


```
func BackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[      job] ctx Done")
            return
        default:
            time.Sleep(2 * time.Second)
            log.Println("[      job] Done")
        }
    }
}

func HeavyBackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()
            log.Println("[heavy job] ctx Done")
            return
        default:
            time.Sleep(15 * time.Second)
            log.Println("[heavy job] Done")
        }
    }
}
```

Graceful shutdown of background processes

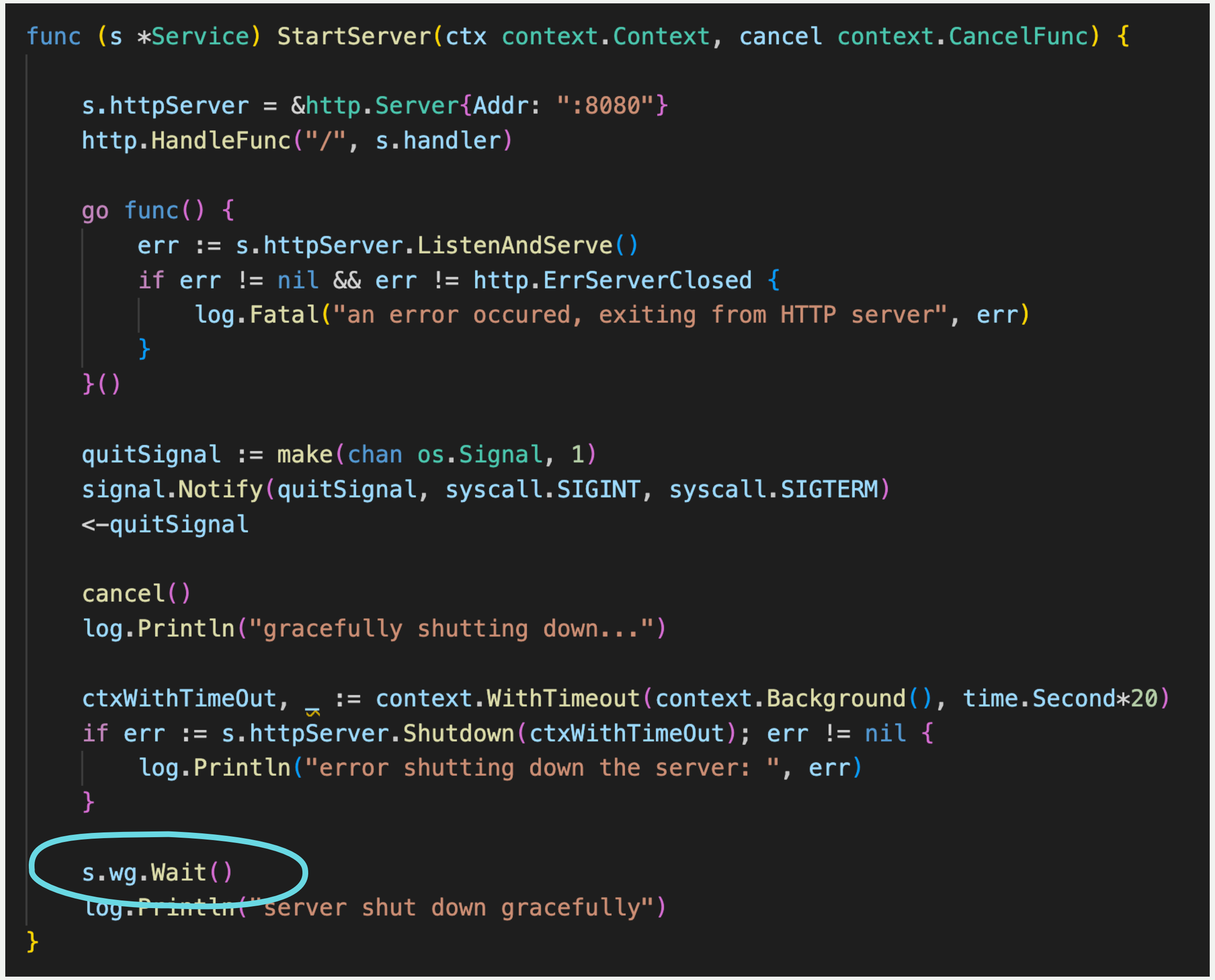


```go
s.wg.Add(1)
go jobs.BackgroundJob(ctx, s.wg)

s.wg.Add(1)
go jobs.HeavyBackgroundJob(ctx, s.wg)
```



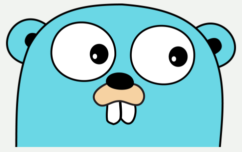
```go
func BackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done()  -1
            log.Println("[      job] ctx Done")
            return
        default:
            time.Sleep(2 * time.Second)
            log.Println("[      job] Done")
        }
    }
}

func HeavyBackgroundJob(ctx context.Context, wg *sync.WaitGroup) {
    for {
        select {
        case <-ctx.Done():
            wg.Done() -1
            log.Println("[heavy job] ctx Done")
            return
        default:
            time.Sleep(15 * time.Second)
            log.Println("[heavy job] Done")
        }
    }
}
```

Wait() blocks until all goroutines finish



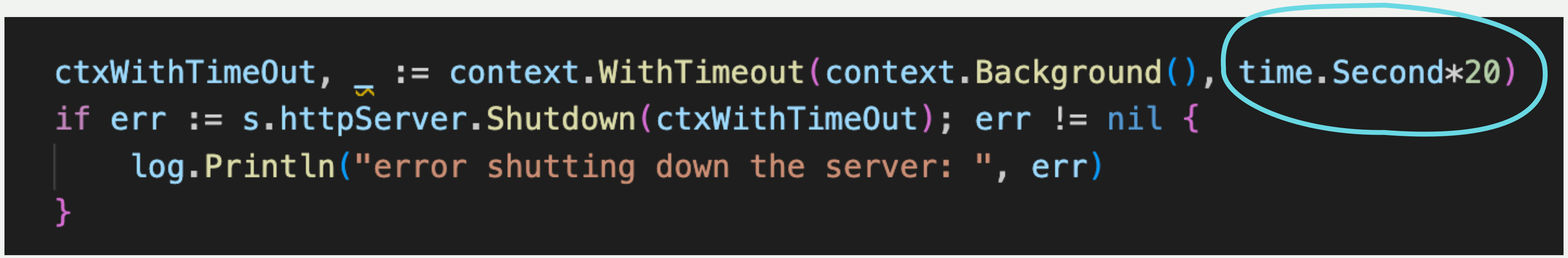
```go
func (s *Service) StartServer(ctx context.Context, cancel context.CancelFunc) {

    s.httpServer = &http.Server{Addr: ":8080"}
    http.HandleFunc("/", s.handler)

    go func() {
        err := s.httpServer.ListenAndServe()
        if err != nil && err != http.ErrServerClosed {
            log.Fatal("an error occured, exiting from HTTP server", err)
        }
    }()

    quitSignal := make(chan os.Signal, 1)
    signal.Notify(quitSignal, syscall.SIGINT, syscall.SIGTERM)
    <-quitSignal

    cancel()
    log.Println("gracefully shutting down...")

    ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
    if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
        log.Println("error shutting down the server: ", err)
    }

    s.wg.Wait()
    log.Println("server shut down gracefully")
}
```

✓ Completion of all background processes



WaitGroups

Graceful shutdown of server



```
ctxWithTimeOut, _ := context.WithTimeout(context.Background(), time.Second*20)
if err := s.httpServer.Shutdown(ctxWithTimeOut); err != nil {
    log.Println("error shutting down the server: ", err)
}
```

- ✓ Completion of requests in process
- ✓ Rejection of new requests

https://twitter.com/ashleymcnamara/status/893580781112700928