# Chronos: Concurrent Job Dispatcher in Go

Chronos is a concurrency-heavy backend system built in Go that simulates a distributed job processing system. It uses goroutines, channels, context, timeouts, retries, synchronization primitives, and graceful shutdown to manage job execution in a robust and production-grade way.

---

## Table of Contents



---

## Overview

Chronos simulates a backend system that:

- Accepts a list of jobs
- Dispatches them with throttling
- Processes them concurrently with worker goroutines
- Applies timeout logic to each job
- Retries failed or timed-out jobs
- Tracks stats for monitoring
- Handles graceful shutdown with Ctrl+C

---

## Concurrency Concepts Used

| Concept | Usage |
|---|---|
| Goroutines | For concurrent worker execution |
| Channels | Job queue and results |

| Concept | Usage |
|---|---|
| Select | Handling timeouts and rate limiting |
| Context | Timeout + graceful cancellation |
| Mutex | Safe stats updates |
| WaitGroup | Waiting for all workers |
| Atomic | Retry counters |
| OS Signals | Handling shutdown |

---

## Project Structure

```
chronos/
├── cmd/
│   └── main.go
├── internal/
│   ├── dispatcher/
│   │   └── dispatcher.go
│   ├── job/
│   │   ├── job.go
│   │   └── processor.go
│   ├── shutdown/
│   │   └── shutdown.go
│   └── worker/
│       └── pool.go
```

---

## Step-by-Step Breakdown

### Step 1: Job Definition (`internal/job/job.go`)

```go
package job // handles job definition and logic

import (
    "fmt"         // for printing job status
    "sync/atomic" // for safe concurrent retry counter updates
)

type Status string // type alias for job status values
```

```
const (
    Pending  Status = "pending"  // job is created but not started
    Running  Status = "running"  // job is currently being processed
    Success  Status = "success"  // job completed successfully
    Failed   Status = "failed"   // job failed during processing
    TimedOut Status = "timed_out" // job exceeded allowed time
)

type Job struct {
    ID      int     // unique job identifier
    Payload string  // content or task to be processed
    Retries int32   // number of retries
    Status  Status  // current status of the job
}

// MarkRetry increments the retry count atomically
func (j *Job) MarkRetry() {
    atomic.AddInt32(&j.Retries, 1)
}

// LogStatus prints current status and retry count
func (j *Job) LogStatus() {
    fmt.Printf("[Job %d] Status: %s | Retries: %d\n", j.ID, j.Status, j.Retries)
}
```

**Step 2: Job Processing (`internal/job/processor.go`)**

```
package job // continues the job logic

import (
    "fmt"       // for logging
    "math/rand" // to introduce random failures and durations
    "time"      // to simulate job duration
)

func Process(j *Job) error {
    fmt.Printf("[Processor] Job #%d starting with payload: %s\n", j.ID,
j.Payload)

    // simulate processing time between 100ms to 1s
    workTime := time.Duration(rand.Intn(900)+100) * time.Millisecond
    time.Sleep(workTime)

    // simulate a 20% chance of job failure
    if rand.Float32() < 0.2 {
        return fmt.Errorf("simulated failure")
```

```
    }

    fmt.Printf("[Processor] Job #%d completed successfully\n", j.ID)
    return nil
}
```

**Step 3: Worker Pool (`internal/worker/pool.go`)**

```
package worker // manages concurrent workers

import (
    "chronos/internal/job"
    "context"
    "fmt"
    "sync"
    "time"
)

// Stats tracks processed and failed jobs safely
type Stats struct {
    Processed int       // number of successful jobs
    Failed    int       // number of failed jobs
    Mutex     sync.Mutex // protects the stats from race conditions
}

func StartWorker(id int, jobs <-chan *job.Job, results chan<- *job.Job, wg
*sync.WaitGroup, stats *Stats) {
    defer wg.Done() // signal when worker is done

    for j := range jobs { // keep picking jobs until channel is closed
        j.Status = job.Running
        ctx, cancel := context.WithTimeout(context.Background(), 1*time.Second)
        resultChan := make(chan error, 1)

        // run the job processing in a goroutine to allow timeout
        go func() {
            resultChan <- job.Process(j)
        }()

        select {
        case <-ctx.Done(): // job took too long
            j.Status = job.TimedOut
            j.MarkRetry()
            stats.Mutex.Lock()
            stats.Failed++
            stats.Mutex.Unlock()
```

```
            fmt.Printf("[Worker %d] Job #%d timed out\n", id, j.ID)

        case err := <-resultChan:
            if err != nil {
                j.Status = job.Failed
                j.MarkRetry()
                stats.Mutex.Lock()
                stats.Failed++
                stats.Mutex.Unlock()
                fmt.Printf("[Worker %d] Job #%d failed: %v\n", id, j.ID, err)
            } else {
                j.Status = job.Success
                stats.Mutex.Lock()
                stats.Processed++
                stats.Mutex.Unlock()
                results <- j
            }
        }
        cancel()
    }
}
```

**Step 4: Dispatcher (`internal/dispatcher/dispatcher.go`)**

```
package dispatcher // handles job dispatch logic

import (
    "chronos/internal/job"
    "context"
    "fmt"
    "time"
)

func StartDispatcher(ctx context.Context, jobs []*job.Job, queue chan *job.Job,
rate <-chan time.Time) {
    for _, j := range jobs {
        select {
        case <-ctx.Done(): // stop dispatching if shutdown is triggered
            fmt.Println("[Dispatcher] Shutdown signal received. Stopping
dispatch...")
            close(queue)
            return
        case <-rate: // wait for rate limiter tick
            fmt.Printf("[Dispatcher] Dispatching Job #%d\n", j.ID)
            queue <- j
        }
```

```
    }
    close(queue) // signal to workers: no more jobs
}
```

**Step 5: Graceful Shutdown (`internal/shutdown/shutdown.go`)**

```
package shutdown // handles OS interrupt signals

import (
    "context"
    "os"
    "os/signal"
    "syscall"
)

func Listen(cancel context.CancelFunc) {
    c := make(chan os.Signal, 1) // channel to capture signals
    signal.Notify(c, os.Interrupt, syscall.SIGTERM) // listen to interrupt/
terminate

    go func() {
        <-c // wait until a signal is received
        println("\n[Shutdown] Caught interrupt. Gracefully stopping...")
        cancel() // cancel the context
    }()
}
```

**Step 6: Main Integration (`cmd/main.go`)**

```
package main

import (
    "chronos/internal/dispatcher"
    "chronos/internal/job"
    "chronos/internal/shutdown"
    "chronos/internal/worker"
    "context"
    "fmt"
    "sync"
    "time"
)

func main() {
    ctx, cancel := context.WithCancel(context.Background()) // create a global
cancel context
```

```
    defer cancel()

    shutdown.Listen(cancel) // setup graceful shutdown on Ctrl+C

    jobQueue := make(chan *job.Job, 10)     // main job queue
    results := make(chan *job.Job, 10)      // results channel for successful
jobs
    rate := time.Tick(300 * time.Millisecond) // throttle rate: 1 job every
300ms
    stats := &worker.Stats{} // shared stats

    // Create sample jobs
    var jobList []*job.Job
    for i := 1; i <= 15; i++ {
        jobList = append(jobList, &job.Job{
            ID:      i,
            Payload: fmt.Sprintf("Payload-%d", i),
            Status:  job.Pending,
        })
    }

    // Start the dispatcher
    go dispatcher.StartDispatcher(ctx, jobList, jobQueue, rate)

    // Launch worker pool
    const numWorkers = 4
    var wg sync.WaitGroup
    for i := 1; i <= numWorkers; i++ {
        wg.Add(1)
        go worker.StartWorker(i, jobQueue, results, &wg, stats)
    }

    wg.Wait()        // wait for all workers to complete
    close(results)   // no more results to receive

    // Print stats
    fmt.Println("\n========== FINAL REPORT ==========")
    fmt.Printf("Processed Jobs : %d\n", stats.Processed)
    fmt.Printf("Failed Jobs    : %d\n", stats.Failed)
    fmt.Println("==================================")
}
```

---

## Final Result

- Jobs are dispatched every 300ms

- Workers handle jobs concurrently
- Failed and timed-out jobs are tracked
- Graceful shutdown happens on Ctrl+C

---

## Next Steps

| Feature | Description |
|---|---|
| REST API | Accept live jobs via HTTP |
| Retry Backoff | Use exponential backoff for retries |
| Persistent Queue | Redis or DB-based retrying system |
| Observability | Logrus for logging, Prometheus for metrics |
| Unit Tests | Interfaces + mock testing |
| Real-World Use | Convert to a microservice with endpoints |