## **1. Overview**

### **1.1 Purpose**

Develop a middleware system in **Go** using the **Ergo (Go)** framework to:

* Connect the desktop application to various package locations.
* Decide optimal locations for item pickups based on proximity and capacity.
* Generate and serve customs documents and bills of lading in HTML format for user access.

### **1.2 Scope**

This system will handle:

* Order processing and allocation to appropriate warehouses.
* Dynamic batching and dispatching based on capacity and urgency.
* Real-time tracking of packages.
* Document generation and delivery to users.

### **1.3 Key Requirements**

* **Concurrency and Scalability:** Efficient handling of numerous simultaneous orders and processes.
* **Fault Tolerance:** Robustness against failures with automatic recovery.
* **Maintainability:** Easy updates and extensions without significant downtime.
* **Halakhic Compliance:** Avoid making promises by providing estimated statuses without fixed commitments.
* **Integration:** Seamless interaction with the existing Go backend and desktop application.

## **2. System Architecture**

### **2.1 Architectural Style**

* **Actor Model:** Utilize Ergo (Go) actors for isolated, concurrent processes.
* **Event-Driven:** Processes communicate via asynchronous message passing.
* **Modular:** Clear separation of concerns among different components.

### **2.2 Components Overview**

1. **API Gateway**
2. **Order Processor**
3. **Warehouse Manager**
4. **Truck Manager**
5. **Dispatcher**
6. **Document Generator**
7. **Tracker**
8. **Supervisor Hierarchy**

## **3. Components and Responsibilities**

### **3.1 API Gateway**

#### **Responsibilities:**

* Serve as the entry point for the desktop application.
* Handle HTTP/HTTPS requests for order submissions, status inquiries, and document retrieval.
* Authenticate and authorize requests.

#### **Endpoints:**

* POST /orders: Submit a new order.
* GET /orders/{id}: Retrieve order status.
* GET /documents/{id}: Fetch generated documents.

#### **Implementation:**

Use a web framework like **Echo** for routing and handling HTTP requests.

go

Copy code

// main.go

func main() {

e := echo.New()

e.POST("/orders", handleNewOrder)

e.GET("/orders/:id", handleGetOrderStatus)

e.GET("/documents/:id", handleGetDocument)

e.Start(":8080")

}

### **3.2 Order Processor**

#### **Responsibilities:**

* Receive new orders from the API Gateway.
* Assign orders to the nearest warehouse with available inventory.
* Communicate with the Warehouse Manager for allocation.

#### **Messages Handled:**

* NewOrder
* OrderAssigned
* OrderFailed

#### **Implementation:**

An Ergo actor that processes incoming orders.

go

Copy code

type OrderProcessor struct{}

func (op \*OrderProcessor) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case NewOrder:

nearestWarehouse := findNearestWarehouse(msg.Order)

if nearestWarehouse != nil {

ctx.Send(nearestWarehouse, AssignOrder{Order: msg.Order})

} else {

ctx.Send(ctx.Self(), OrderFailed{OrderID: msg.Order.ID, Reason: "No inventory available"})

}

case OrderAssigned:

// Update order status and notify user if necessary

case OrderFailed:

// Handle order failure (e.g., notify user)

}

}

### **3.3 Warehouse Manager**

#### **Responsibilities:**

* Manage inventory for a specific warehouse.
* Allocate items to orders.
* Track current load and capacity.

#### **Messages Handled:**

* AssignOrder
* UpdateInventory
* CheckCapacity

#### **Implementation:**

Separate actors for each warehouse.

go

Copy code

type WarehouseManager struct {

ID string

Inventory map[string]int

MaxCapacity int

CurrentLoad int

}

func (wm \*WarehouseManager) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case AssignOrder:

if wm.Inventory[msg.Order.Item] > 0 {

wm.Inventory[msg.Order.Item]--

wm.CurrentLoad += 1

ctx.Send(ctx.Parent(), OrderAssigned{OrderID: msg.Order.ID, WarehouseID: wm.ID})

} else {

ctx.Send(ctx.Parent(), OrderFailed{OrderID: msg.Order.ID, Reason: "Item out of stock"})

}

case CheckCapacity:

ctx.Send(ctx.Parent(), CapacityStatus{WarehouseID: wm.ID, CurrentLoad: wm.CurrentLoad, MaxCapacity: wm.MaxCapacity})

}

}

### **3.4 Truck Manager**

#### **Responsibilities:**

* Track the capacity and current load of each truck.
* Decide when to dispatch based on capacity or urgency.
* Assign shipments to trucks.

#### **Messages Handled:**

* AddShipment
* DispatchTruck
* CheckDispatch

#### **Implementation:**

Separate actors for each truck.

go

Copy code

type TruckManager struct {

ID string

MaxLoad int

CurrentLoad int

Route string

}

func (tm \*TruckManager) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case AddShipment:

tm.CurrentLoad += msg.Weight

if tm.CurrentLoad >= int(0.7\*float64(tm.MaxLoad)) || msg.Urgent {

ctx.Send(ctx.Self(), DispatchTruck{})

}

case DispatchTruck:

dispatchTruck(tm.Route, tm.CurrentLoad)

tm.CurrentLoad = 0

}

}

### **3.5 Dispatcher**

#### **Responsibilities:**

* Oversee all trucks and initiate dispatching based on conditions.
* Optimize routes and load distribution.

#### **Messages Handled:**

* CheckDispatch
* TruckReady
* DispatchDecision

#### **Implementation:**

An Ergo actor that periodically checks truck statuses.

go

Copy code

type Dispatcher struct{}

func (d \*Dispatcher) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case CheckDispatch:

for \_, truck := range getAllTrucks() {

ctx.Send(truck, CheckCapacity{})

}

case CapacityStatus:

if msg.CurrentLoad >= int(0.7\*float64(msg.MaxCapacity)) {

ctx.Send(getTruckManager(msg.WarehouseID), DispatchTruck{})

}

}

}

### **3.6 Document Generator**

#### **Responsibilities:**

* Generate HTML documents for bills of lading and customs.
* Serve or store generated documents for user access.
* Optionally convert HTML to PDF for download.

#### **Messages Handled:**

* GenerateBillOfLading
* GenerateCustomsDocument

#### **Implementation:**

An Ergo actor that handles document generation.

go

Copy code

type DocumentGenerator struct{}

func (dg \*DocumentGenerator) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case GenerateBillOfLading:

html := generateHTMLDocument(msg.OrderID)

storeDocument(msg.OrderID, "bill\_of\_lading.html", html)

case GenerateCustomsDocument:

html := generateCustomsDocument(msg.OrderID)

storeDocument(msg.OrderID, "customs\_document.html", html)

}

}

### **3.7 Tracker**

#### **Responsibilities:**

* Monitor real-time locations of packages.
* Update statuses based on GPS or other tracking mechanisms.
* Communicate updates to relevant actors and the API Gateway.

#### **Messages Handled:**

* UpdateLocation
* GetPackageStatus

#### **Implementation:**

An Ergo actor that handles tracking.

go

Copy code

type Tracker struct {

PackageLocations map[string]string // PackageID -> Location

}

func (t \*Tracker) HandleMessage(ctx ergo.Context, message interface{}) {

switch msg := message.(type) {

case UpdateLocation:

t.PackageLocations[msg.PackageID] = msg.Location

ctx.Send(ctx.Parent(), LocationUpdated{PackageID: msg.PackageID, Location: msg.Location})

case GetPackageStatus:

location, exists := t.PackageLocations[msg.PackageID]

if exists {

ctx.Send(ctx.Sender(), PackageStatus{PackageID: msg.PackageID, Location: location})

} else {

ctx.Send(ctx.Sender(), PackageStatus{PackageID: msg.PackageID, Location: "Unknown"})

}

}

}

### **3.8 Supervisor Hierarchy**

#### **Responsibilities:**

* Monitor and manage actors.
* Restart actors in case of failures.
* Ensure system reliability and fault tolerance.

#### **Implementation:**

Use Ergo’s supervision features to define a supervision tree.

func main() {

supervisor := ergo.Supervisor{

// Define supervision strategies and children actors

}

supervisor.Start()

}

## **4. Data Models**

### **4.1 Order**

type Order struct {

ID string

CustomerID string

Item string

Quantity int

Destination string

Priority string // e.g., "urgent", "standard"

Status string // e.g., "pending", "assigned", "dispatched"

CreatedAt time.Time

UpdatedAt time.Time

}

### **4.2 Shipment**

type Shipment struct {

ID string

OrderID string

WarehouseID string

TruckID string

Status string // e.g., "loaded", "in transit", "delivered"

DispatchedAt time.Time

}

### **4.3 Warehouse**

type Warehouse struct {

ID string

Location string

Inventory map[string]int

MaxCapacity int

CurrentLoad int

}

### **4.4 Truck/Boat/Plane**

type Truck struct {

ID string

Route string

MaxLoad int

CurrentLoad int

Status string // e.g., "idle", "loading", "dispatched"

Location string

}

### **4.5 Document**

type Document struct {

ID string

OrderID string

Type string // e.g., "bill\_of\_lading", "customs"

Content string // HTML content

GeneratedAt time.Time

}

## **5. Interaction Diagrams**

### **5.1 Order Processing Workflow**

1. **Order Submission:**
   * User submits an order via the desktop app.
   * API Gateway receives the POST /orders request.
   * API Gateway sends a NewOrder message to the OrderProcessor actor.
2. **Order Allocation:**
   * OrderProcessor finds the nearest WarehouseManager.
   * Sends AssignOrder message to the selected WarehouseManager.
3. **Inventory Check:**
   * WarehouseManager checks inventory.
   * If available, decrements inventory and sends OrderAssigned to OrderProcessor.
   * If unavailable, sends OrderFailed to OrderProcessor.
4. **Update Order Status:**
   * OrderProcessor updates the order status accordingly.
   * Notifies the user via the API Gateway.
5. **Shipment Assignment:**
   * If assigned, OrderProcessor communicates with TruckManager to add the shipment.
   * TruckManager decides to dispatch based on current load and urgency.
6. **Dispatching:**
   * TruckManager dispatches the truck.
   * Updates shipment status to "dispatched."
7. **Document Generation:**
   * Upon dispatch, Dispatcher or TruckManager sends GenerateBillOfLading and GenerateCustomsDocument messages to DocumentGenerator.
   * DocumentGenerator creates HTML documents and stores them.
8. **User Access:**
   * User can retrieve documents via GET /documents/{id} endpoint.

### **5.2 Real-Time Tracking Workflow**

1. **Location Update:**
   * GPS devices send location updates to the middleware.
   * API Gateway receives location data and sends UpdateLocation messages to Tracker.
2. **Status Update:**
   * Tracker updates the package location.
   * Notifies relevant actors and updates order status.
3. **User Inquiry:**
   * User requests package status via GET /orders/{id}.
   * API Gateway retrieves status from OrderProcessor or directly from Tracker.

## **6. Message Definitions**

### **6.1 NewOrder**

type NewOrder struct {

Order Order

}

### **6.2 AssignOrder**

type AssignOrder struct {

Order Order

}

### **6.3 OrderAssigned**

type OrderAssigned struct {

OrderID string

WarehouseID string

}

### **6.4 OrderFailed**

type OrderFailed struct {

OrderID string

Reason string

}

### **6.5 AddShipment**

type AddShipment struct {

Shipment Shipment

Urgent bool

}

### **6.6 DispatchTruck**

type DispatchTruck struct{}

### **6.7 GenerateBillOfLading**

type GenerateBillOfLading struct {

OrderID string

}

### **6.8 GenerateCustomsDocument**

type GenerateCustomsDocument struct {

OrderID string

}

### **6.9 UpdateLocation**

type UpdateLocation struct {

PackageID string

Location string

}

### **6.10 GetPackageStatus**

type GetPackageStatus struct {

PackageID string

}

### **6.11 PackageStatus**

type PackageStatus struct {

PackageID string

Location string

}

## **7. Document Generation**

### **7.1 HTML Templates**

Create HTML templates for bills of lading and customs documents using Go’s html/template package.

**Example: templates/bill\_of\_lading.html**

html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Bill of Lading</title>

</head>

<body>

<h1>Bill of Lading</h1>

<p><strong>Order ID:</strong> {{.OrderID}}</p>

<p><strong>Shipper:</strong> {{.Shipper}}</p>

<p><strong>Consignee:</strong> {{.Consignee}}</p>

<p><strong>Items:</strong></p>

<ul>

{{range .Items}}

<li>{{.Name}} - Quantity: {{.Quantity}}</li>

{{end}}

</ul>

<p><strong>Delivery Date:</strong> {{.DeliveryDate}}</p>

</body>

</html>

### **7.2 Document Generation Logic**

Use Go’s html/template to populate templates with order data.

func generateHTMLDocument(orderID string) string {

order := getOrderByID(orderID)

tmpl, err := template.ParseFiles("templates/bill\_of\_lading.html")

if err != nil {

log.Fatalf("Template parsing error: %v", err)

}

var htmlContent strings.Builder

err = tmpl.Execute(&htmlContent, order)

if err != nil {

log.Fatalf("Template execution error: %v", err)

}

return htmlContent.String()

}

### **7.3 Storing and Serving Documents**

Store generated documents in a storage service (e.g., AWS S3, local filesystem) and provide URLs for retrieval.

func storeDocument(orderID, docType, content string) {

filename := fmt.Sprintf("%s\_%s.html", orderID, docType)

filepath := path.Join("documents", filename)

err := os.WriteFile(filepath, []byte(content), 0644)

if err != nil {

log.Fatalf("Failed to store document: %v", err)

}

}

### **7.4 Optional: Convert HTML to PDF**

Use a library like **Chromedp** to convert HTML documents to PDF.

import (

"context"

"github.com/chromedp/chromedp"

"io/ioutil"

)

func generatePDF(htmlPath, pdfPath string) error {

ctx, cancel := chromedp.NewContext(context.Background())

defer cancel()

var buf []byte

err := chromedp.Run(ctx, chromedp.Tasks{

chromedp.Navigate("file://" + htmlPath),

chromedp.ActionFunc(func(ctx context.Context) error {

var err error

buf, \_, err = page.PrintToPDF().Do(ctx)

return err

}),

})

if err != nil {

return err

}

return ioutil.WriteFile(pdfPath, buf, 0644)

}

## **8. API Specifications**

### **8.1 Submit a New Order**

* **Endpoint:** POST /orders

**Request Body:**  
{

"customer\_id": "C123",

"item": "Teddy Bear",

"quantity": 2,

"destination": "Edwardsville",

"priority": "standard"

}

**Response:**  
{

"order\_id": "O456",

"status": "pending"

}

### **8.2 Get Order Status**

* **Endpoint:** GET /orders/{id}

**Response:**  
{

"order\_id": "O456",

"status": "assigned",

"warehouse\_id": "W789",

"current\_location": "St. Louis Warehouse",

"delivery\_estimate": "2024-01-15"

}

### **8.3 Get Documents**

* **Endpoint:** GET /documents/{id}
* **Response:**
  + Serve the HTML document directly.
  + Optionally provide a link to download the PDF version.

## **9. Error Handling and Supervision**

### **9.1 Supervisor Hierarchy**

Define a supervision tree where each primary actor (e.g., OrderProcessor, Dispatcher) is supervised. If an actor fails, the supervisor can restart it based on defined strategies.

func setupSupervisors() {

supervisor := ergo.Supervisor{

// Define children and their supervision strategies

}

supervisor.Start()

}

### **9.2 Error Messages**

Define specific messages for error handling, such as OrderFailed, DispatchFailed, etc., and ensure actors handle these gracefully by retrying or logging appropriately.

## **10. Deployment and Updates**

### **10.1 Deployment Strategy**

* **Containerization:** Use Docker to containerize the middleware for consistent deployments.
* **Orchestration:** Deploy using Kubernetes or similar for scalability and management.
* **CI/CD Pipeline:** Automate testing and deployment using tools like GitHub Actions, Jenkins, or GitLab CI.

### **10.2 Code Updates**

Since hot code updates are not required:

* **Immutable Actors:** Existing actors run their initial code version. New actors spawn with updated code upon deployment.
* **Rolling Updates:** Deploy updates incrementally, restarting services to use the latest code without downtime.
* **Graceful Shutdowns:** Ensure actors finish processing current messages before shutting down to avoid data loss.

## **11. Security Considerations**

### **11.1 Authentication and Authorization**

* Implement secure authentication (e.g., OAuth 2.0) for API endpoints.
* Ensure only authorized users can access or modify orders and documents.

### **11.2 Data Protection**

* **Encryption:** Use HTTPS for all communications. Encrypt sensitive data at rest and in transit.
* **Input Validation:** Validate all incoming data to prevent injection attacks and ensure data integrity.

### **11.3 Access Controls**

* Define roles and permissions to restrict access to different parts of the system based on user roles.

## **12. Testing Strategy**

### **12.1 Unit Testing**

* Test individual actors’ message handling logic.
* Mock dependencies to isolate tests.

### **12.2 Integration Testing**

* Test interactions between multiple actors (e.g., OrderProcessor and WarehouseManager).
* Ensure correct message flows and state changes.

### **12.3 End-to-End Testing**

* Simulate real-world scenarios from order submission to document retrieval.
* Verify system behavior under various conditions (e.g., high load, failures).

### **12.4 Automated Testing**

* Integrate tests into the CI/CD pipeline for continuous validation.

## **13. Monitoring and Logging**

### **13.1 Monitoring**

* Use monitoring tools like **Prometheus** and **Grafana** to track system metrics (e.g., number of active actors, message rates, error rates).
* Set up alerts for critical issues (e.g., actor failures, high latency).

### **13.2 Logging**

* Implement structured logging using libraries like **Logrus** or **Zap**.
* Log key events, errors, and state changes for debugging and auditing purposes.

## **14. Scalability Considerations**

### **14.1 Horizontal Scaling**

* Scale actors across multiple instances or nodes as load increases.
* Use a distributed actor system if necessary to handle high concurrency.

### **14.2 Load Balancing**

* Distribute incoming requests evenly across multiple instances of the API Gateway.
* Ensure stateful actors (e.g., WarehouseManager, TruckManager) are correctly partitioned or replicated.

### **14.3 Resource Management**

* Monitor and optimize resource usage (CPU, memory) to prevent bottlenecks.
* Implement rate limiting to handle sudden spikes in traffic gracefully.

## **15. Data Persistence**

### **15.1 Database Integration**

* Use a reliable database (e.g., PostgreSQL, MongoDB) to store persistent data like orders, shipments, warehouses, and documents.
* Ensure data consistency and integrity with proper transactions and constraints.

### **15.2 Data Models Persistence**

Map Go data models to database schemas, ensuring efficient queries and updates.

go

Copy code

// Example using GORM

type Order struct {

ID string `gorm:"primaryKey"`

CustomerID string

Item string

Quantity int

Destination string

Priority string

Status string

CreatedAt time.Time

UpdatedAt time.Time

}

func saveOrder(order Order) error {

return db.Create(&order).Error

}

func getOrderByID(id string) Order {

var order Order

db.First(&order, "id = ?", id)

return order

}

## **16. Example Workflows**

### **16.1 Order Submission and Processing**

1. **User Action:** Submits an order via the desktop app.
2. **API Gateway:** Receives POST /orders and creates a NewOrder message.
3. **OrderProcessor:** Receives NewOrder, finds the nearest WarehouseManager, and sends AssignOrder.
4. **WarehouseManager:** Allocates inventory, updates status, and sends OrderAssigned.
5. **OrderProcessor:** Updates order status and notifies the user.
6. **TruckManager:** Adds shipment and decides on dispatching based on capacity or urgency.
7. **Dispatcher:** Monitors trucks and initiates dispatch.
8. **DocumentGenerator:** Creates necessary documents and stores them for user access.

### **16.2 Real-Time Tracking**

1. **GPS Device:** Sends location updates to the middleware.
2. **API Gateway:** Receives location data and sends UpdateLocation to Tracker.
3. **Tracker:** Updates package location and notifies relevant actors.
4. **User Inquiry:** Requests package status via GET /orders/{id}.
5. **API Gateway:** Retrieves status from Tracker and returns it to the user.

## **17. Deployment Diagram**

sql

Copy code

+--------------------+ +----------------------+ +-----------------------+

| Desktop App | <------> | API Gateway | <------> | Middleware |

+--------------------+ +----------------------+ +-----------------------+

| | | |

v v v v

+-----------+ +----------+ +---------+

| Order | | Dispatcher| | Tracker |

| Processor | | | | |

+-----------+ +----------+ +---------+

| |

v v

+--------------+ +--------------+

| Warehouse | | TruckManager |

| Managers | +--------------+

+--------------+

|

v

+----------------+

| Document |

| Generator |

+----------------+

## **18. Conclusion**

This **Code Specification** provides a detailed blueprint for developing a robust, scalable, and maintainable middleware system using **Go** and **Ergo (Go)**. By leveraging the actor model, event-driven architecture, and Go’s concurrency strengths, the system efficiently manages orders, optimizes logistics, and generates necessary documentation while ensuring fault tolerance and ease of maintenance. The design also respects halakhic principles by avoiding fixed promises and focusing on dynamic, efficient operations.

## **19. Next Steps**

1. **Prototype Development:**
   * Start by implementing core actors like OrderProcessor and WarehouseManager.
   * Develop basic API endpoints and test end-to-end workflows.
2. **Iterative Testing:**
   * Perform unit and integration tests to ensure each component functions correctly.
   * Simulate real-world scenarios to validate system behavior.
3. **Scaling and Optimization:**
   * Optimize actor interactions and message flows.
   * Implement monitoring and logging to track system performance.
4. **Deployment:**
   * Containerize the application and set up CI/CD pipelines.
   * Deploy to a staging environment for further testing before production rollout.
5. **Documentation and Training:**
   * Document the system architecture, APIs, and deployment processes.
   * Train the development and operations teams on managing and maintaining the system.

By following this specification, you can systematically build a middleware system that meets your business needs, ensures reliability, and scales with your logistics operations.