ecommerce-platform / order-service (Step 3)

This is the third service of the project: **order-service** (orders + payments) implemented in Go with gRPC, Postgres, and Docker. It depends on **product-service** to check and deduct stock before creating an order.

Features

- gRPC service with **CreateOrder**, **GetOrder**, **ListOrders**
- Calls `product-service.UpdateStock` via gRPC before creating orders
- Postgres storage (`orders` table + order items)
- Basic validation & error handling
- Health + reflection
- Proto + Makefile target to generate stubs
- Dockerfile
- Updated `docker-compose.yml` (adds this service + DB)

Directory tree

```
ecommerce-platform/
└── order-service/
    ├── Dockerfile
    ├── Makefile
    ├── go.mod
    ├── go.sum
    ├── main.go
    ├── proto/
    │   └── order.proto
    └── internal/
        ├── config/
        │   └── config.go
        ├── db/
        │   └── postgres.go
        ├── models/
        │   └── order.go
        ├── client/
        │   └── product_client.go
        └── service/
            └── order_server.go
```

---

proto/order.proto

```
syntax = "proto3";
package order.v1;

option go_package = "github.com/example/ecommerce-platform/order-service/
proto;orderpb";

message OrderItem {
  string product_id = 1;
  int32 quantity = 2;
  double price = 3;
}

message Order {
  string id = 1;
  string user_id = 2;
  repeated OrderItem items = 3;
  double total = 4;
  string status = 5; // e.g. CREATED, PAID, SHIPPED
  string created_at = 6; // RFC3339
}

message CreateOrderRequest {
  string user_id = 1;
  repeated OrderItem items = 2;
}
message CreateOrderResponse { Order order = 1; }

message GetOrderRequest { string id = 1; }
message GetOrderResponse { Order order = 1; }

message ListOrdersRequest { string user_id = 1; }
message ListOrdersResponse { repeated Order orders = 1; }

service OrderService {
  rpc CreateOrder(CreateOrderRequest) returns (CreateOrderResponse);
  rpc GetOrder(GetOrderRequest) returns (GetOrderResponse);
  rpc ListOrders(ListOrdersRequest) returns (ListOrdersResponse);
}
```

---

order-service/go.mod

```
module github.com/example/ecommerce-platform/order-service

go 1.22.0
```

```
require (
    github.com/jackc/pgx/v5 v5.6.0
    github.com/jackc/pgx/v5/stdlib v5.6.0
    google.golang.org/grpc v1.65.0
    google.golang.org/grpc/health v1.1.0
    google.golang.org/protobuf v1.34.2
)
```

---

order-service/internal/config/config.go

```go
package config

import (
    "log"
    "os"
)

type Config struct {
    Port        string
    DatabaseURL string
    ProductAddr string
}

func FromEnv() Config {
    cfg := Config{
        Port:        getEnv("PORT", "50053"),
        DatabaseURL: getEnv("DATABASE_URL", "postgres://app:app@localhost:
5432/orders?sslmode=disable"),
        ProductAddr: getEnv("PRODUCT_ADDR", "product-service:50052"),
    }
    if cfg.DatabaseURL == "" {
        log.Fatal("DATABASE_URL must be set")
    }
    return cfg
}

func getEnv(key, def string) string {
    if v := os.Getenv(key); v != "" {
        return v
    }
    return def
}
```

---

order-service/internal/db/postgres.go

```
package db

import (
    "context"
    "database/sql"
    "errors"
    "time"

    _ "github.com/jackc/pgx/v5/stdlib"
)

type Postgres struct { DB *sql.DB }

func Connect(dsn string) (*Postgres, error) {
    database, err := sql.Open("pgx", dsn)
    if err != nil { return nil, err }
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()
    if err := database.PingContext(ctx); err != nil { return nil, err }
    p := &Postgres{DB: database}
    if err := p.migrate(ctx); err != nil { return nil, err }
    return p, nil
}

func (p *Postgres) migrate(ctx context.Context) error {
    stmt := `
    CREATE TABLE IF NOT EXISTS orders (
        id TEXT PRIMARY KEY,
        user_id TEXT NOT NULL,
        total DOUBLE PRECISION NOT NULL,
        status TEXT NOT NULL,
        created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
    );
    CREATE TABLE IF NOT EXISTS order_items (
        order_id TEXT REFERENCES orders(id) ON DELETE CASCADE,
        product_id TEXT NOT NULL,
        quantity INT NOT NULL,
        price DOUBLE PRECISION NOT NULL
    );
    `
    _, err := p.DB.ExecContext(ctx, stmt)
    return err
}

var ErrNotFound = errors.New("not found")

// Simplified queries (omitting joins for brevity)
```

order-service/internal/client/product_client.go

```
package client

import (
    "context"

    "github.com/example/ecommerce-platform/product-service/proto"
    "google.golang.org/grpc"
)

type ProductClient struct {
    c proto.ProductServiceClient
}

func NewProductClient(addr string) (*ProductClient, error) {
    conn, err := grpc.Dial(addr, grpc.WithInsecure())
    if err != nil { return nil, err }
    return &ProductClient{c: proto.NewProductServiceClient(conn)}, nil
}

func (pc *ProductClient) UpdateStock(ctx context.Context, productID string,
delta int32) error {
    _, err := pc.c.UpdateStock(ctx, &proto.UpdateStockRequest{Id: productID,
Delta: delta})
    return err
}
```

order-service/internal/service/order_server.go

```
package service

import (
    "context"
    "time"

    "github.com/google/uuid"
    "github.com/example/ecommerce-platform/order-service/internal/client"
    "github.com/example/ecommerce-platform/order-service/internal/db"
    "github.com/example/ecommerce-platform/order-service/proto"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
)

type OrderServer struct {
    proto.UnimplementedOrderServiceServer
```

```
    db     *db.Postgres
    pc     *client.ProductClient
}

func NewOrderServer(database *db.Postgres, pc *client.ProductClient)
*OrderServer {
    return &OrderServer{db: database, pc: pc}
}

func (s *OrderServer) CreateOrder(ctx context.Context, req
*proto.CreateOrderRequest) (*proto.CreateOrderResponse, error) {
    if len(req.Items) == 0 {
        return nil, status.Error(codes.InvalidArgument, "no items")
    }
    var total float64
    for _, item := range req.Items {
        if err := s.pc.UpdateStock(ctx, item.ProductId, -item.Quantity);
err != nil {
            return nil, status.Errorf(codes.FailedPrecondition, "stock: %v",
err)
        }
        total += float64(item.Quantity) * item.Price
    }
    id := uuid.NewString()
    // Insert into DB (simplified)
    _, err := s.db.DB.ExecContext(ctx,
        `INSERT INTO orders (id,user_id,total,status) VALUES ($1,$2,$3,$4)`,
        id, req.UserId, total, "CREATED",
    )
    if err != nil { return nil, status.Errorf(codes.Internal, "db: %v",
err) }
    for _, item := range req.Items {
        _, _ = s.db.DB.ExecContext(ctx,
            `INSERT INTO order_items (order_id,product_id,quantity,price)
VALUES ($1,$2,$3,$4)`,
            id, item.ProductId, item.Quantity, item.Price,
        )
    }
    order := &proto.Order{
        Id: id,
        UserId: req.UserId,
        Items: req.Items,
        Total: total,
        Status: "CREATED",
        CreatedAt: time.Now().UTC().Format(time.RFC3339),
    }
    return &proto.CreateOrderResponse{Order: order}, nil
}

func (s *OrderServer) GetOrder(ctx context.Context, req
*proto.GetOrderRequest) (*proto.GetOrderResponse, error) {
```

```
    row := s.db.DB.QueryRowContext(ctx, `SELECT
user_id,total,status,created_at FROM orders WHERE id=$1`, req.Id)
    var userID, statusStr string
    var total float64
    var createdAt time.Time
    if err := row.Scan(&userID, &total, &statusStr, &createdAt); err != nil {
        return nil, status.Error(codes.NotFound, "order not found")
    }
    order := &proto.Order{Id: req.Id, UserId: userID, Total: total, Status:
statusStr, CreatedAt: createdAt.Format(time.RFC3339)}
    return &proto.GetOrderResponse{Order: order}, nil
}

func (s *OrderServer) ListOrders(ctx context.Context, req
*proto.ListOrdersRequest) (*proto.ListOrdersResponse, error) {
    rows, err := s.db.DB.QueryContext(ctx,
`SELECT id,total,status,created_at FROM orders WHERE user_id=$1`, req.UserId)
    if err != nil { return nil, status.Errorf(codes.Internal, "db: %v",
err) }
    defer rows.Close()
    var orders []*proto.Order
    for rows.Next() {
        var id, statusStr string
        var total float64
        var createdAt time.Time
        if err := rows.Scan(&id,&total,&statusStr,&createdAt); err != nil {
return nil, err }
        orders = append(orders, &proto.Order{
            Id: id, UserId: req.UserId, Total: total, Status: statusStr,
CreatedAt: createdAt.Format(time.RFC3339),
        })
    }
    return &proto.ListOrdersResponse{Orders: orders}, nil
}
```

---

order-service/main.go

```
package main

import (
    "log"
    "net"

    "github.com/example/ecommerce-platform/order-service/internal/client"
    "github.com/example/ecommerce-platform/order-service/internal/config"
    "github.com/example/ecommerce-platform/order-service/internal/db"
    "github.com/example/ecommerce-platform/order-service/internal/service"
    "github.com/example/ecommerce-platform/order-service/proto"
```

```
    "google.golang.org/grpc"
    "google.golang.org/grpc/health"
    hpb "google.golang.org/grpc/health/grpc_health_v1"
    "google.golang.org/grpc/reflection"
)

func main() {
    cfg := config.FromEnv()

    pg, err := db.Connect(cfg.DatabaseURL)
    if err != nil { log.Fatalf("db connect: %v", err) }
    defer pg.DB.Close()

    pc, err := client.NewProductClient(cfg.ProductAddr)
    if err != nil { log.Fatalf("product client: %v", err) }

    lis, err := net.Listen("tcp", ":"+cfg.Port)
    if err != nil { log.Fatalf("listen: %v", err) }

    gsrv := grpc.NewServer()
    proto.RegisterOrderServiceServer(gsrv, service.NewOrderServer(pg, pc))

    h := health.NewServer()
    hpb.RegisterHealthServer(gsrv, h)
    reflection.Register(gsrv)

    log.Printf("order-service listening on %s", lis.Addr().String())
    if err := gsrv.Serve(lis); err != nil {
        log.Fatalf("serve: %v", err)
    }
}
```

---

order-service/Dockerfile

```
FROM golang:1.22 AS builder
WORKDIR /app

COPY go.mod go.sum ./
RUN go mod download

COPY . .
RUN CGO_ENABLED=0 GOOS=linux go build -o /order-service ./main.go

FROM gcr.io/distroless/base-debian12
COPY --from=builder /order-service /order-service
EXPOSE 50053
ENTRYPOINT ["/order-service"]
```

Updated deployments/docker-compose.yml

```
version: "3.9"
services:
  db_orders:
    image: postgres:16
    container_name: orders_db
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app
      POSTGRES_DB: orders
    ports:
      - "5434:5432"
    volumes:
      - orders-data:/var/lib/postgresql/data

  order-service:
    build: ../order-service
    container_name: order_service
    environment:
      DATABASE_URL: postgres://app:app@db_orders:5432/orders?sslmode=disable
      PORT: "50053"
      PRODUCT_ADDR: product-service:50052
    depends_on:
      - db_orders
      - product-service
    ports:
      - "50053:50053"

volumes:
  orders-data: {}
```

How to run (this step)

```
# Generate protobuf code
cd ecommerce-platform/order-service
make proto
go mod tidy

# Run stack
cd ../deployments
docker compose up --build

# Test via grpcurl
# CreateOrder
grpcurl -plaintext -d '{"user_id":"U1","items":
```

```
[{"product_id":"P1","quantity":1,"price":10.5}]}'
  localhost:50053 order.v1.OrderService/CreateOrder
```

This completes Step 3: a working `order-service` integrated with `product-service`.