ecommerce-platform / user-service (Step 1)

This is the first service of the project: **user-service** (auth + user profiles) implemented in Go with gRPC, Postgres, JWT, and Docker. It includes:

- gRPC service with **Register**, **Login**, **GetProfile**
- JWT-based auth (HS256)
- Postgres storage (email unique, bcrypt password hashing)
- Health + reflection
- Proto + Makefile target to generate stubs
- Dockerfile
- Minimal `docker-compose.yml` (for DB + this service only)

Directory tree (for this step)

```
ecommerce-platform/
└── user-service/
    ├── Dockerfile
    ├── Makefile
    ├── go.mod
    ├── go.sum                # generated after `go mod tidy`
    ├── main.go
    ├── proto/
    │   └── user.proto
    └── internal/
        ├── auth/
        │   └── jwt.go
        ├── config/
        │   └── config.go
        ├── db/
        │   └── postgres.go
        ├── hash/
        │   └── password.go
        ├── models/
        │   └── user.go
        └── service/
            └── user_server.go
```

---

proto/user.proto

```proto
syntax = "proto3";
package user.v1;
```

```
option go_package = "github.com/example/ecommerce-platform/user-service/
proto;userpb";

message User {
  string id = 1;
  string email = 2;
  string name = 3;
  string created_at = 4; // RFC3339
}

message RegisterRequest {
  string email = 1;
  string password = 2;
  string name = 3;
}

message AuthResponse {
  User user = 1;
  string access_token = 2;
}

message LoginRequest {
  string email = 1;
  string password = 2;
}

message GetProfileRequest {
  string id = 1; // optional; if empty, service will read from JWT `sub`
}

message GetProfileResponse {
  User user = 1;
}

service UserService {
  rpc Register(RegisterRequest) returns (AuthResponse);
  rpc Login(LoginRequest) returns (AuthResponse);
  rpc GetProfile(GetProfileRequest) returns (GetProfileResponse);
}
```

---

user-service/Makefile

```
PROTO_DIR=proto

proto:
    protoc
        --go_out=paths=source_relative:.
        --go-grpc_out=paths=source_relative:.
```

```
        $(PROTO_DIR)/*.proto

run:
    go run ./...

build:
    go build -o bin/user-service ./...

clean:
    rm -rf bin
```

---

user-service/go.mod

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

go 1.22.0

require (
    github.com/golang-jwt/jwt/v5 v5.2.1
    github.com/jackc/pgx/v5 v5.6.0
    github.com/jackc/pgx/v5/stdlib v5.6.0
    golang.org/x/crypto v0.28.0
    google.golang.org/grpc v1.65.0
    google.golang.org/grpc/health v1.1.0
    google.golang.org/protobuf v1.34.2
)
```

After pasting files, run `go mod tidy` to resolve versions.

---

user-service/internal/config/config.go

```
package config

import (
    "log"
    "os"
)

type Config struct {
    Port       string
    DatabaseURL string
    JWTSecret  string
}

func FromEnv() Config {
```

```
    cfg := Config{
        Port:       getEnv("PORT", "50051"),
        DatabaseURL: getEnv("DATABASE_URL", "postgres://app:app@localhost:
5432/users?sslmode=disable"),
        JWTSecret:  os.Getenv("JWT_SECRET"),
    }
    if cfg.JWTSecret == "" {
        log.Fatal("JWT_SECRET must be set")
    }
    return cfg
}

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

---

user-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
    }
    database.SetMaxOpenConns(10)
    database.SetMaxIdleConns(5)
    database.SetConnMaxLifetime(30 * time.Minute)

    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 {
    // Minimal schema; `id` managed by app
    stmt := `
    CREATE TABLE IF NOT EXISTS users (
        id TEXT PRIMARY KEY,
        email TEXT NOT NULL UNIQUE,
        password_hash TEXT NOT NULL,
        name TEXT,
        created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
    );
    `
    _, err := p.DB.ExecContext(ctx, stmt)
    return err
}

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

// ----- Queries -----

type UserRow struct {
    ID           string
    Email        string
    PasswordHash string
    Name         sql.NullString
    CreatedAt    time.Time
}

func (p *Postgres) GetUserByEmail(ctx context.Context, email string)
(*UserRow, error) {
    row := p.DB.QueryRowContext(ctx, `SELECT id, email, password_hash, name,
created_at FROM users WHERE email=$1`, email)
    var u UserRow
    if err := row.Scan(&u.ID, &u.Email, &u.PasswordHash, &u.Name,
&u.CreatedAt); err != nil {
        if errors.Is(err, sql.ErrNoRows) { return nil, ErrNotFound }
        return nil, err
    }
    return &u, nil
}

func (p *Postgres) GetUserByID(ctx context.Context, id string) (*UserRow,
error) {
```

```
    row := p.DB.QueryRowContext(ctx, `SELECT id, email, password_hash, name,
created_at FROM users WHERE id=$1`, id)
    var u UserRow
    if err := row.Scan(&u.ID, &u.Email, &u.PasswordHash, &u.Name,
&u.CreatedAt); err != nil {
        if errors.Is(err, sql.ErrNoRows) { return nil, ErrNotFound }
        return nil, err
    }
    return &u, nil
}

func (p *Postgres) CreateUser(ctx context.Context, id, email, passwordHash,
name string) (*UserRow, error) {
    _, err := p.DB.ExecContext(ctx,
        `INSERT INTO users (id, email, password_hash, name) VALUES
($1,$2,$3,$4)`,
        id, email, passwordHash, nullable(name),
    )
    if err != nil {
        return nil, err
    }
    return p.GetUserByID(ctx, id)
}

func nullable(s string) any {
    if s == "" { return nil }
    return s
}
```

---

user-service/internal/hash/password.go

```
package hash

import "golang.org/x/crypto/bcrypt"

func HashPassword(pw string) (string, error) {
    b, err := bcrypt.GenerateFromPassword([]byte(pw), bcrypt.DefaultCost)
    return string(b), err
}

func CheckPassword(hash, pw string) error {
    return bcrypt.CompareHashAndPassword([]byte(hash), []byte(pw))
}
```

> If your Go version doesn't have `ComparePasswordAndHash`, use `bcrypt.CompareHashAndPassword([]byte(hash), []byte(pw))`.

Replace the function body with:

```
return bcrypt.CompareHashAndPassword([]byte(hash), []byte(pw))
```

---

user-service/internal/auth/jwt.go

```
package auth

import (
    "time"

    "github.com/golang-jwt/jwt/v5"
)

type TokenMaker struct { secret []byte }

func NewTokenMaker(secret string) *TokenMaker { return &TokenMaker{secret:
[]byte(secret)} }

func (t *TokenMaker) Generate(userID, email string, ttl time.Duration)
(string, error) {
    claims := jwt.RegisteredClaims{
        Subject:   userID,
        ExpiresAt: jwt.NewNumericDate(time.Now().Add(ttl)),
        IssuedAt:  jwt.NewNumericDate(time.Now()),
    }
    tok := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return tok.SignedString(t.secret)
}

func (t *TokenMaker) Parse(tokenStr string) (*jwt.RegisteredClaims, error) {
    parser := jwt.NewParser()
    t, err := parser.ParseWithClaims(tokenStr, &jwt.RegisteredClaims{},
func(token *jwt.Token) (interface{}, error) {
        return t.secret, nil
    })
    if err != nil { return nil, err }
    if claims, ok := t.Claims(t); ok {
        _ = claims // unreachable; keeping to avoid confusion
    }
    claims, ok := t.Claims.(*jwt.RegisteredClaims)
    if !ok { return nil, jwt.ErrTokenInvalidClaims }
    return claims, nil
}
```

Note: Some editors complain about the odd `Claims` usage. A safer approach:

```
func (t *TokenMaker) Parse(tokenStr string) (*jwt.RegisteredClaims, error) {
    token, err := jwt.ParseWithClaims(tokenStr, &jwt.RegisteredClaims{},
func(token *jwt.Token) (interface{}, error) {
        return t.secret, nil
    })
    if err != nil { return nil, err }
    claims, ok := token.Claims.(*jwt.RegisteredClaims)
    if !ok || !token.Valid { return nil, jwt.ErrTokenInvalidClaims }
    return claims, nil
}
```

---

user-service/internal/models/user.go

```
package models

type User struct {
    ID        string
    Email     string
    Name      string
    CreatedAt string // RFC3339
}
```

---

user-service/internal/service/user_server.go

```
package service

import (
    "context"
    "fmt"
    "strings"
    "time"

    "github.com/google/uuid"
    "github.com/example/ecommerce-platform/user-service/internal/auth"
    "github.com/example/ecommerce-platform/user-service/internal/db"
    "github.com/example/ecommerce-platform/user-service/internal/hash"
    "github.com/example/ecommerce-platform/user-service/internal/models"
    "github.com/example/ecommerce-platform/user-service/proto"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/metadata"
    "google.golang.org/grpc/status"
)

type UserServer struct {
```

```
    proto.UnimplementedUserServiceServer
    db    *db.Postgres
    token *auth.TokenMaker
}

func NewUserServer(database *db.Postgres, token *auth.TokenMaker)
*UserServer {
    return &UserServer{db: database, token: token}
}

func (s *UserServer) Register(ctx context.Context, req
*proto.RegisterRequest) (*proto.AuthResponse, error) {
    if req.GetEmail() == "" || req.GetPassword() == "" {
        return nil, status.Error(codes.InvalidArgument, "email and password
required")
    }
    if _, err := s.db.GetUserByEmail(ctx, req.Email); err == nil {
        return nil, status.Error(codes.AlreadyExists, "email already
registered")
    }
    h, err := hash.HashPassword(req.Password)
    if err != nil { return nil, status.Errorf(codes.Internal, "hash: %v",
err) }
    id := uuid.NewString()
    row, err := s.db.CreateUser(ctx, id, strings.ToLower(req.Email), h,
req.Name)
    if err != nil { return nil, status.Errorf(codes.Internal, "create user:
%v", err) }
    tok, err := s.token.Generate(row.ID, row.Email, 24*time.Hour)
    if err != nil { return nil, status.Errorf(codes.Internal, "token: %v",
err) }
    return &proto.AuthResponse{User: toProto(row), AccessToken: tok}, nil
}

func (s *UserServer) Login(ctx context.Context, req *proto.LoginRequest)
(*proto.AuthResponse, error) {
    row, err := s.db.GetUserByEmail(ctx, strings.ToLower(req.Email))
    if err != nil {
        return nil, status.Error(codes.NotFound, "invalid credentials")
    }
    if err := hash.CheckPassword(row.PasswordHash, req.Password); err != nil
{
        return nil, status.Error(codes.Unauthenticated, "invalid
credentials")
    }
    tok, err := s.token.Generate(row.ID, row.Email, 24*time.Hour)
    if err != nil { return nil, status.Errorf(codes.Internal, "token: %v",
err) }
    return &proto.AuthResponse{User: toProto(row), AccessToken: tok}, nil
}
```

```
func (s *UserServer) GetProfile(ctx context.Context, req
*proto.GetProfileRequest) (*proto.GetProfileResponse, error) {
    userID := req.GetId()
    if userID == "" {
        idFromJWT, err := s.userIDFromContext(ctx)
        if err != nil { return nil, status.Error(codes.Unauthenticated,
"missing/invalid token") }
```