Folder structures:

```
 Folder PATH listing for volume OS
Volume serial number is F4F3-F694
C:.
│   .air.toml
│   .env
│   docker-compose.yml
│   Dockerfile
│   go.mod
│   go.sum
│
├───cmd
│   │   main.go
│   │
│   └───migrate
│       └───migrations
│               000001_create_users.down.sql
│               000001_create_users.up.sql
│               000002_create_friends.down.sql
│               000002_create_friends.up.sql
│               000003_create_posts.down.sql
│               000003_create_posts.up.sql
│               000004_create_comments.down.sql
│               000004_create_comments.up.sql
│               000005_create_messages.down.sql
│               000005_create_messages.up.sql
│
├───infrastructure
│   ├───cache
│   │       redis.go
│   │
│   └───db
│           connect.go
│           migrate.go
│
├───internal
│   ├───api
│   │       routes.go
│   │
│   ├───entity
│   │       comment.go
│   │       friend.go
│   │       friend_recommendation.go
│   │       message.go
│   │       post.go
│   │       user.go
│   │
│   ├───handler
│   │       chat_handler.go
│   │       comment_handler.go
│   │       friend_handler.go
│   │       post_handler.go
│   │       user_handler.go
│   │
│   ├───middleware
│   │       auth_middleware.go
│   │       middleware.go
│   │
│   ├───model
│   │   ├───request
│   │   │       chat_request.go
│   │   │       comment_request.go
│   │   │       friend_request.go
│   │   │       mood_prediction_request.go
```

```
│   │   │       post_request.go
│   │   │       user_request.go
│   │   │
│   │   └──response
│   │           chat_response.go
│   │           comment_response.go
│   │           friend_response.go
│   │           mood_prediction_response.go
│   │           post_response.go
│   │           user_response.go
│   │
│   ├──repository
│   │       chat_repository.go
│   │       comment_repository.go
│   │       friend_repository.go
│   │       post_repository.go
│   │       user_repository.go
│   │
│   ├──service
│   │       auth_service.go
│   │       chat_service.go
│   │       comment_service.go
│   │       friend_service.go
│   │       mood_prediction_service.go
│   │       post_service.go
│   │       user_service.go
│   │       websocket_client.go
│   │       websocket_hub.go
│   │
│   └──utils
│           comment_validation.go
│           friend_validation.go
│           post_validation.go
│           user_validation.go
│
└──tmp
        build-errors.log
```

Database Design:

```
CREATE TABLE IF NOT EXISTS users (
    UserID BIGSERIAL PRIMARY KEY,
    Username VARCHAR(50) NOT NULL UNIQUE,
    Fullname VARCHAR(50) NOT NULL,
    ProfileUrl VARCHAR(255),
    Email VARCHAR(50) NOT NULL UNIQUE,
    Password VARCHAR(255) NOT NULL,
    CreatedAt TIMESTAMP DEFAULT NOW()
);

CREATE TABLE IF NOT EXISTS friends (
    FriendID SERIAL PRIMARY KEY,
    UserID INTEGER REFERENCES users(UserID) ON DELETE CASCADE,
    FriendUserID INTEGER REFERENCES users(UserID) ON DELETE CASCADE,
    FriendStatus BOOLEAN DEFAULT 'PENDING',
    CreatedAt TIMESTAMP DEFAULT NOW()
);

CREATE TABLE IF NOT EXISTS posts (
    PostID SERIAL PRIMARY KEY,
    UserID INTEGER REFERENCES users(UserID) ON DELETE CASCADE,
    Content TEXT NOT NULL,
    Mood VARCHAR(50) DEFAULT 'NORMAL',
    CreatedAt TIMESTAMP DEFAULT NOW()
);

CREATE TABLE IF NOT EXISTS comments (
    CommentID SERIAL PRIMARY KEY,
    PostID INTEGER REFERENCES posts(PostID) ON DELETE CASCADE,
    UserID INTEGER REFERENCES users(UserID) ON DELETE CASCADE,
    Content TEXT NOT NULL,
    CreatedAt TIMESTAMP DEFAULT NOW()
);

CREATE TABLE IF NOT EXISTS messages (
    MessageID BIGSERIAL PRIMARY KEY,
    SenderID BIGINT REFERENCES users(UserID) ON DELETE CASCADE,
    RecipientID BIGINT REFERENCES users(UserID) ON DELETE CASCADE,
    Content TEXT NOT NULL,
    Timestamp TIMESTAMP DEFAULT NOW(),
    Status VARCHAR(50) DEFAULT 'sent' -- Possible values: 'sent', 'delivered', 'read', 'failed' [cite: 50]
);
```

main.go:

```
package main

import (
    "mood-bridge-v2/server/infrastructure/cache"
    "mood-bridge-v2/server/infrastructure/db"
    "mood-bridge-v2/server/internal/api"
)

func main() {
    // Pertama-tama kita akan inisialisasi koneksi ke database PostgreSQL
    database := db.NewDbConnection()
    defer database.Close()

    // Setelah itu kita akan migrasi database-nya
    db.Migrate(database, "up")

    // Setup redis client untuk caching
    rdb := cache.NewRedisClient()
    defer rdb.Close()

    // Terakhir kita jalankan server-nya
    router := api.SetupRoutes(database, rdb)
    router.Run(":8080")
}
```

api/routes.go

```
package api

import (
    "database/sql"
    "mood-bridge-v2/server/internal/handler"
    "mood-bridge-v2/server/internal/middleware"
    "mood-bridge-v2/server/internal/repository"
    "mood-bridge-v2/server/internal/service"

    "github.com/gin-gonic/gin"
    "github.com/go-playground/validator/v10"
    "github.com/redis/go-redis/v9"
)

// step 1: define sebuah Handler sebagai struct yang akan digunakan untuk mengumpulkan semua handler yang ada dalam rest api kita.
type Handlers struct {
    UserHandler    handler.UserHandler
    PostHandler    handler.PostHandler
    CommentHandler handler.CommentHandler
    FriendHandler handler.FriendHandler
    ChatHandler    handler.ChatHandler
}

// step 2: buat method untuk setiap route yang ada dalam api kita. misal kita mau bikin route untuk create user, kita bisa bikin met
func SetupRoutes(db *sql.DB, redisClient *redis.Client) *gin.Engine {
    return initRoutes(initHandler(db, redisClient))
}

func initHandler(db *sql.DB, redisClient *redis.Client) Handlers {
    // Inisialisasi validator juga
    validator := validator.New()

    // Inisialisasi repository, handler, dan services disini
    userRepository := repository.NewUserRepository()
    userService := service.NewUserService(db, userRepository)
    userHandler := handler.NewUserHandler(userService, *validator)
```

```
    postRepository := repository.NewPostRepository()
    postService := service.NewPostService(db, postRepository, userRepository, service.NewMoodPredictionService(), redisClient)
    postHandler := handler.NewPostHandler(postService, *validator)

    commentRepository := repository.NewCommentRepository()
    commentService := service.NewCommentService(commentRepository, userRepository, postRepository, db, redisClient)
    commentHandler := handler.NewCommentHandler(commentService, *validator)

    friendRepository := repository.NewFriendRepository()
    friendService := service.NewFriendService(friendRepository, userRepository, db, redisClient)
    friendHandler := handler.NewFriendHandler(friendService, *validator)

    chatRepository := repository.NewChatRepository(db)
    websocketHub := service.NewConcreteHub(chatRepository)
    chatService := service.NewChatService(chatRepository, websocketHub)
    chatHandler := handler.NewChatHandler(chatService)

    return Handlers{
        UserHandler:    userHandler,
        PostHandler:    postHandler,
        CommentHandler: commentHandler,
        FriendHandler: friendHandler,
        ChatHandler:    chatHandler,
    }
}

func initRoutes(h Handlers) *gin.Engine {
    // Inisialisasi router
    router := gin.Default()

    // Terapkan middleware untuk menangani panic
    router.Use(middleware.HandlePanic())
    // Terapkan middleware untuk CORS
    router.Use(middleware.CORSMiddleware())

    // Lakukan grouping
    api := router.Group("/api")

    // Buat routes untuk user
    user := api.Group("/user")
    {
        user.POST("/register", h.UserHandler.Create)
        user.POST("/login", h.UserHandler.Login)
        user.GET("/by-username/:username", h.UserHandler.Find)
        user.GET("/by-id/:id", h.UserHandler.FindByID)

        user.Use(middleware.Authenticate())
        user.GET("/by-email", h.UserHandler.FindByEmail)
        user.GET("/all", h.UserHandler.FindAll)
        user.PUT("/update/:id", h.UserHandler.Update)
    }

    post := api.Group("/post")
    {
        post.GET("/all", h.PostHandler.FindAll)
        post.GET("/by-id/:id", h.PostHandler.Find)
        post.GET("/by-userid/:id", h.PostHandler.FindByUserID)

        post.Use(middleware.Authenticate())
        post.POST("/create", h.PostHandler.Create)
        post.PUT("/update/:id", h.PostHandler.Update)
        post.DELETE("/delete/:id", h.PostHandler.Delete)
        post.GET("/friend-posts/:id", h.PostHandler.GetFriendPosts)
    }
```

```
    comment := api.Group("/comment")
    {
        comment.GET("/by-postid/:id", h.CommentHandler.GetAllByPostID)
        comment.GET("/by-id/:id", h.CommentHandler.GetByID)

        comment.Use(middleware.Authenticate())
        comment.POST("/create", h.CommentHandler.Create)
        comment.DELETE("/delete/:id", h.CommentHandler.Delete)
    }

    friend := api.Group("/friend")
    {
        friend.GET("/all/:id", h.FriendHandler.GetFriends)
        friend.GET("/requests/:id", h.FriendHandler.GetFriendRequests)
        friend.Use(middleware.Authenticate())
        friend.POST("/add", h.FriendHandler.AddFriend)
        friend.POST("/accept", h.FriendHandler.AcceptRequest)
        friend.DELETE("/delete/:id", h.FriendHandler.Delete)
        friend.GET("/recommendation/:id", h.FriendHandler.GetFriendRecommendation)
    }

    chat := api.Group("/chat")
    {
        chat.Use(middleware.Authenticate()) // gaboleh dituker
        chat.GET("/ws", h.ChatHandler.HandleWebSocketConnection)
        chat.GET("/history", h.ChatHandler.HandleFetchChatHistory)
        chat.POST("/messages/:message_id/read", h.ChatHandler.HandleMarkMessageAsRead)
    }

    return router
}
```

entity/user.go

```
package entity

import (
    "database/sql"
    "time"
)

type User struct {
    ID         int            `gorm:"primaryKey;autoIncrement"`
    Username   string         `gorm:"type:varchar(100);unique;not null" json:"username"`
    Fullname   string         `gorm:"type:varchar(100);not null" json:"fullname"`
    Email      string         `gorm:"type:varchar(100);unique;not null" json:"email"`
    Password   string         `gorm:"type:varchar(255);not null" json:"-"`
    ProfileUrl sql.NullString `gorm:"type:varchar(255)" json:"profileurl"`
    Posts      []Post         `gorm:"foreignKey:UserID"`
    CreatedAt  time.Time      `json:"createdat"`
}
```

entity/post.go

```
package entity

import "time"

type Post struct {
    PostID    int       `gorm:"primaryKey;autoIncrement"`
    UserID    int       `gorm:"not null" json:"user_id"`
    User      User      `gorm:"foreignKey:UserID" json:"user"` // Ini perlu biar bisa db.Preload("User").Find(&posts)
    Content   string    `json:"content"`
    Mood      string    `json:"mood"`
    CreatedAt time.Time `json:"created_at"`
}
```

entity/comment.go

```
package entity

import "time"

type Comment struct {
    CommentID int       `gorm:"primaryKey;autoIncrement" json:"comment_id"`
    UserID    int       `gorm:"not null" json:"user_id"`
    User      User      `gorm:"foreignKey:UserID" json:"user"` // Relasi ke User
    PostID    int       `gorm:"not null" json:"post_id"`
    Post      Post      `gorm:"foreignKey:PostID" json:"post"` // Relasi ke Post
    Content   string    `json:"content"`
    CreatedAt time.Time `json:"created_at"`
}
```

entity/friend.go

```
package entity

import "time"

type Friend struct {
    FriendID      int        `gorm:"primaryKey;autoIncrement" json:"id"`
    UserID        int        `gorm:"not null" json:"userid"`             // orang yang menambahkan teman
    FriendUserID  int        `gorm:"not null" json:"frienduserid"`       // orang yang jadi temannya
    FriendStatus  bool       `gorm:"default:false" json:"friendstatus"` // status teman
    CreatedAt     time.Time  `gorm:"autoCreateTime" json:"createdat"`
    User         *User   `gorm:"foreignKey:FriendUserID;references:ID"`
}

// tar kita bisa nyari temen dengan cara

/*
    db.Preload("User").Preload("FriendUser").Find(&friends)
*/
```

entity/message.go

```
package entity

import "time"

type MessageStatus string

const (
    StatusSent      MessageStatus = "sent"
    StatusDelivered MessageStatus = "delivered"
    StatusRead      MessageStatus = "read"
    StatusFailed    MessageStatus = "failed"
)

type Message struct {
    ID          int   `gorm:"primaryKey;autoIncrement" json:"id"`
    SenderID    int   `json:"senderid"`
    RecipientID int   `json:"recipientid"`
    Content     string   `json:"content"`
    Timestamp   time.Time  `json:"timestamp"`
    Status MessageStatus  `json:"status,omitempty"`
}

// function ini ibaratnya constructor untuk membuat message baru
func NewMessage(senderID, recipientID int, content string) *Message {
    return &Message{
        SenderID:   senderID,
        RecipientID: recipientID,
        Content:    content,
        Timestamp:  time.Now(),
        Status:     StatusSent,
    }
}
```

middleware/auth_middleware.go

```
package middleware

import (
    "context"
    "fmt"
    "log"
    "mood-bridge-v2/server/internal/model/response"
    "net/http"
    "os"
    "strings"

    "github.com/gin-gonic/gin"
    "github.com/golang-jwt/jwt/v5"
    "github.com/joho/godotenv"
)

type Claims struct {
    User *response.CreateUserResponse `json:"user"`
    jwt.RegisteredClaims
}

func Authenticate() gin.HandlerFunc {
    return func(c *gin.Context) {
        // step 1: ambil header authorization dari request
        authHeader := c.GetHeader("authorization")
        if authHeader == "" {
            c.JSON(http.StatusUnauthorized, gin.H{
                "code": http.StatusUnauthorized,
                "message": "Unauthorized - Authorization header missing",
```

```
            })
            c.Abort()
            return
        }

        // step 2: validate token (sekalian ambil claims untuk digunakan sebagai data user)
        parsedClaims, err := ValidateToken(authHeader)
        if err != nil {
            log.Printf("Token validation error: %v\n", err)
            c.JSON(http.StatusUnauthorized, gin.H{
                "code":    http.StatusUnauthorized,
                "message": fmt.Sprintf("Unauthorized - %s", err.Error()),
            })
            c.Abort()
            return
        }

        // step 3: Simpan claims ke gin context, agar bisa diakses di handler apa saja.
        ctx := c.Request.Context() // inisalisasi context-nya
        if parsedClaims.User != nil {
            ctx = context.WithValue(ctx, "username", parsedClaims.User.Username) // simpan username ke context
            if parsedClaims.User.UserID != 0 {
                ctx = context.WithValue(ctx, "userID", parsedClaims.User.UserID) // simpan userID ke context
            } else {
                // seandainya userID tidak ada di token, biasanya error ini terjadi kalau token dibuat sebelum userID ditambahkan ke
                log.Println("UserID is missing or zero in token claims for user:", parsedClaims.User.Username)
                c.JSON(http.StatusUnauthorized, gin.H{
                    "code":   http.StatusUnauthorized,
                    "message": "Unauthorized - User identifier missing in token",
                })
                c.Abort()
                return
            }
        } else {
            // seandainya claims tidak ada user-nya, biasanya error ini terjadi kalau token dibuat sebelum user ditambahkan ke claim
            log.Println("User claim is nil in token for authorization header:", authHeader)
            c.JSON(http.StatusUnauthorized, gin.H{
                "code":    http.StatusUnauthorized,
                "message": "Unauthorized - User data missing in token claims",
            })
            c.Abort()
            return
        }

        // step 4: set context ke request, agar bisa diakses di handler
        c.Request = c.Request.WithContext(ctx)

        // step 5: simpan token ke context untuk digunakan di handler
        c.Set("token", authHeader)

        // step 6: lanjutkan ke handler berikutnya
        c.Next()
    }
}

func ValidateToken(authHeader string) (*Claims, error) {
    // step 1: pastiin token diawali dengan "Bearer "
    if !strings.HasPrefix(authHeader, "Bearer ") {
        return nil, fmt.Errorf("authorization header format must be Bearer {token}")
    }

    // step 2: ambil token dari header
    tokenString := strings.TrimPrefix(authHeader, "Bearer ")
    if tokenString == "" {
        return nil, fmt.Errorf("token string is empty")
    }
```

```
    }

    // step 3: load JWT secret key dari .env file
    err := godotenv.Load()
    if err != nil {
        err = godotenv.Load()
        if err != nil {
            log.Println("Warning: .env file not found, relying on environment variables")
        }
    }
    jwtSecretKey := os.Getenv("JWT_SECRET_KEY")
    if jwtSecretKey == "" {
        log.Println("FATAL: JWT_SECRET_KEY environment variable is not set")
        return nil, fmt.Errorf("server configuration error: JWT_SECRET_KEY is not set")
    }

    // step 4: parse token menjadi claims dengan mengubah ke bentuk []byte dimana []byte adalah tipe data yang dibutuhkan oleh jwt.P
    jwtSecret := []byte(jwtSecretKey)

    // step 5: buat claims untuk menyimpan data user
    claims := &Claims{}
    jwtToken, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (interface{}, error) {
        // step 6: pastikan token menggunakan algoritma HS256
        if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
            return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
        }
        return jwtSecret, nil
    })

    // step 7: jika ada error saat parsing token (termasuk jika token expired), kembalikan error (ERROR HANDLING)
    if err != nil {
        log.Printf("Token parsing error: %v\n", err)
        if err == jwt.ErrTokenExpired {
            return nil, fmt.Errorf("token expired")
        }
        return nil, fmt.Errorf("invalid token: %v", err)
    }

    // step 8: jika token tidak valid, kembalikan error
    if !jwtToken.Valid {
        return nil, fmt.Errorf("token is invalid")
    }

    // step 9: jika semua langkah berhasil, kembalikan claims yang berisi data user (tambah validasi sedikit)
    if claims.User == nil {
        return nil, fmt.Errorf("user data missing in token claims")
    }
    if claims.User.Username == "" {
        return nil, fmt.Errorf("username missing in token claims")
    }

    return claims, nil
}
```

middleware/middleware.go

```
package middleware

import (
    "log"
    "net/http"

    "github.com/gin-gonic/gin"
)

// Middleware buat menangani panic (error yang tidak terduga) di server
// Ini akan mengembalikan response JSON dengan status 500 Internal Server Error
func HandlePanic() gin.HandlerFunc {
    return func(c *gin.Context) {
        // rollback function yang akan dieksekusi
        defer func() {
            r := recover()
            if r != nil {
                log.Println("Panic occurred:", r)
                c.JSON(http.StatusInternalServerError, gin.H{
                    "status": "error",
                    "message": "Internal server error",
                })
                c.Abort()
            }
        }()

        // lanjutkan ke handler berikutnya
        // Jika ada panic di handler berikutnya, maka defer function di atas akan dieksekusi
        c.Next()
    }
}

// Intinya biar localhost:8080 bisa diakses sama localhost:3000
func CORSMiddleware() gin.HandlerFunc {
    return func(c *gin.Context) {
        c.Writer.Header().Set("Access-Control-Allow-Origin", "http://localhost:3000")
        c.Writer.Header().Set("Access-Control-Allow-Credentials", "true")
        c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type, Authorization")
        c.Writer.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")

        if c.Request.Method == "OPTIONS" {
            c.AbortWithStatus(204)
            return
        }

        c.Next()
    }
}
```

model/request/chat_request.go

```
package request

type PrivateMessagePayload struct {
    RecipientID int    `json:"recipientid" binding:"required"`
    Content     string `json:"content" binding:"required,max=1024"`
}

type MarkAsReadPayload struct {
    MessageID int `json:"messageid" binding:"required"`
}
```

model/response/chat_response.go

```
package response

import (
    "mood-bridge-v2/server/internal/entity"
    "time"
)

type ChatMessage struct {
    ID          int    `json:"id"`
    SenderID    int    `json:"senderid"`
    RecipientID int    `json:"recipientid"`
    Content     string    `json:"content"`
    Timestamp   time.Time `json:"timestamp"`
    Status entity.MessageStatus `json:"status,omitempty"`
}

type WebSocketMessage struct {
    Type string   `json:"type"`
    Payload interface{} `json:"payload"` // ini payload isinya bisa berupa ChatMessage, ErrorMessage, dll
}

type ErrorMessage struct {
    Code string `json:"code"`
    Message string `json:"message"`
}
```

repository/chat_repository.go

```
package repository

import (
    "context"
    "database/sql"
    "fmt"
    "mood-bridge-v2/server/internal/entity"
    "time"
)

type ChatRepository interface {
    SaveMessage(ctx context.Context, msg *entity.Message) error
    GetMessagesForConversation(ctx context.Context, senderID, recipientID, limit, offset int) ([]*entity.Message, error)
    UpdateMessageStatus(ctx context.Context, messageID int, newStatus entity.MessageStatus) error
    GetUnreadMessagesForUser(ctx context.Context, userID int, afterTimestamp time.Time) ([]*entity.Message, error)
}

type ChatRepositoryImpl struct {
    DB *sql.DB // Tujuan ini ditaro disini adalah agar DB tidak harus ada dalam hub, client, dsbnya tapi cukup ada pada saat inisial
}

func NewChatRepository(db *sql.DB) ChatRepository {
    return &ChatRepositoryImpl{
        DB: db,
    }
}

func (r *ChatRepositoryImpl) SaveMessage(ctx context.Context, msg *entity.Message) error {
    // step 1: define query buat masukin pesan yang ada ke dalam database
    query := `INSERT INTO messages (senderid, recipientid, content, timestamp, status) VALUES ($1, $2, $3, $4, $5)`

    // step 2: jalankan query-nya (ExecContext) digunakan untuk eksekusi query yang tidak mengembalikan baris (INSERT, UPDATE, DELET
    _, err := r.DB.ExecContext(ctx, query, msg.SenderID, msg.RecipientID, msg.Content, msg.Timestamp, msg.Status)
    if err != nil {
        return fmt.Errorf("error saving message: %w", err)
    }
```

```
    // step 3: jika berhasil, return nil
    return nil
}

func (r *ChatRepositoryImpl) GetMessagesForConversation(ctx context.Context, senderID, recipientID, limit, offset int) ([]*entity.Me
    // step 1: define query untuk mengambil pesan dari database baik yang sudah di read maupun yang belum di read
    query := `
    SELECT id, senderid, recipientid, content, timestamp, status
    FROM messages
    WHERE (senderid = $1 AND recipientid = $2) OR (senderid = $2 AND recipientid = $1)
    ORDER BY timestamp ASC
    LIMIT $3 OFFSET $4
    `

    // step 2: jalankan query-nya (QueryContext) digunakan untuk eksekusi query yang mengembalikan banyak baris (SELECT)
    rows, err := r.DB.QueryContext(ctx, query, senderID, recipientID, limit, offset)
    if err != nil {
        return nil, fmt.Errorf("error retrieving messages: %w", err)
    }

    // step 3: pastikan untuk menutup rows setelah selesai digunakan
    defer rows.Close()

    // step 4: buat slice untuk menyimpan pesan yang diambil dari database
    var messages []*entity.Message

    // step 5: iterasi melalui rows untuk mengambil setiap pesan dan scan ke dalam struct entity.Message
    for rows.Next() {
        var msg entity.Message
        if err := rows.Scan(&msg.ID, &msg.SenderID, &msg.RecipientID, &msg.Content, &msg.Timestamp, &msg.Status); err != nil {
            fmt.Printf("error scanning message: %v\n", err)
            continue
        }
        messages = append(messages, &msg)
    }

    // step 6: periksa apakah ada error saat iterasi
    if err = rows.Err(); err != nil {
        return nil, fmt.Errorf("error iterating over messages: %w", err)
    }

    // step 7: jika berhasil, return slice of messages
    return messages, nil
}

func (r *ChatRepositoryImpl) UpdateMessageStatus(ctx context.Context, messageID int, newStatus entity.MessageStatus) error {
    // step 1: define query untuk update status pesan
    query := `UPDATE messages SET status = $1 WHERE id = $2`

    // step 2: jalankan query-nya (ExecContext) digunakan untuk eksekusi query yang tidak mengembalikan baris (UPDATE)
    result, err := r.DB.ExecContext(ctx, query, newStatus, messageID)
    if err != nil {
        return fmt.Errorf("error updating message status for %d: %w", messageID, err)
    }

    // step 3: cek apakah ada baris yang terpengaruh (RowsAffected) untuk memastikan pesan dengan ID tersebut ada
    rowsAffected, err := result.RowsAffected()
    if err != nil {
        return fmt.Errorf("error getting rows affected for message %d: %w", messageID, err)
    }

    // step 4: jika tidak ada baris yang terpengaruh, berarti pesan dengan ID tersebut tidak ditemukan
    if rowsAffected == 0 {
        return fmt.Errorf("no message found with ID %d to update status", messageID)
    }
```

```
    }

    // step 5: jika berhasil, return nil
    return nil
}

func (r *ChatRepositoryImpl) GetUnreadMessagesForUser(ctx context.Context, userID int, afterTimestamp time.Time) ([]*entity.Message,
    // step 1: define query untuk mengambil pesan yang statusnya bukan read dan setelah timestamp tertentu
    query := `
    SELECT messageid, senderid, recipientid, content, timestamp, status
    FROM messages
    WHERE (recipientid = $1 AND status != $2) AND timestamp > $3
    ORDER BY timestamp ASC
    `

    // step 2: jalankan query-nya (QueryContext) digunakan untuk eksekusi query yang mengembalikan banyak baris (SELECT)
    rows, err := r.DB.QueryContext(ctx, query, userID, entity.StatusRead, afterTimestamp)
    if err != nil {
        return nil, fmt.Errorf("error retrieving unread messages for user %d: %w", userID, err)
    }

    // step 3: pastikan untuk menutup rows setelah selesai digunakan
    defer rows.Close()

    // step 4: buat slice untuk menyimpan pesan yang diambil dari database
    var messages []*entity.Message
    for rows.Next() {
        var msg entity.Message
        if err := rows.Scan(&msg.ID, &msg.SenderID, &msg.RecipientID, &msg.Content, &msg.Timestamp, &msg.Status); err != nil {
            fmt.Printf("error scanning unread message: %v\n", err)
            continue
        }
        messages = append(messages, &msg)
    }

    // step 5: periksa apakah ada error saat iterasi
    if err = rows.Err(); err != nil {
        return nil, fmt.Errorf("error iterating over unread messages for user %d: %w", userID, err)
    }

    // step 6: jika berhasil, return slice of unread messages
    return messages, nil
}
```

service/chat_service.go

```
package service

import (
    "context"
    "encoding/json"
    "errors"
    "fmt"
    "log"
    "mood-bridge-v2/server/internal/entity"
    "mood-bridge-v2/server/internal/model/response"
    "mood-bridge-v2/server/internal/repository"
    "time"

    "github.com/gorilla/websocket"
)

type ChatService interface {
    HandleNewConnection(ctx context.Context, userID int, conn *websocket.Conn) error
```

```
    HandleIncomingMessage(ctx context.Context, senderID int, recipientID int, content string) error
    FetchConversationHistory(ctx context.Context, senderID, recipientID, limit, offset int) ([]*response.ChatMessage, error)
    MarkMessageAsRead(ctx context.Context, messageID, userID int) error
}

type ChatServiceImpl struct {
    messageRepo repository.ChatRepository
    hub Hub
}

func NewChatService(msgRepo repository.ChatRepository, hub Hub) ChatService {
    return &ChatServiceImpl{
        messageRepo: msgRepo,
        hub: hub,
    }
}

func (s *ChatServiceImpl) HandleNewConnection(ctx context.Context, userID int, conn *websocket.Conn) error {
    // step 1: buat client baru
    client := NewClient(userID, s.hub, conn, s)

    // step 2: hubungkan client ke hub
    s.hub.RegisterClient(client)
    log.Printf("ChatService: User %d connected. Client registered with Hub.", userID)

    // step 3: dapatkan pesan yang belum dibaca
    unreadMessages, err := s.messageRepo.GetUnreadMessagesForUser(ctx, userID, time.Now().Add(-7*24*time.Hour))
    if err != nil {
        log.Printf("ChatService: Error fetching unread messages for user %d: %v", userID, err)
    }

    // step 4: jika pesan yang belum dibaca ditemukan,
    if len(unreadMessages) > 0 {
        log.Printf("ChatService: Sending %d unread messages to user %d", len(unreadMessages), userID)
        // kirim pesan tak terbaca tersebut ke client melalui WebSocket
        for _, msg := range unreadMessages {
            wsMsg := response.WebSocketMessage{
                Type: "offline_message",
                Payload: response.ChatMessage{
                    ID: msg.ID,
                    SenderID: msg.SenderID,
                    RecipientID: msg.RecipientID,
                    Content: msg.Content,
                    Timestamp: msg.Timestamp,
                    Status: msg.Status,
                },
            }

            // ubah pesan ke format JSON
            payloadBytes, err := json.Marshal(wsMsg)
            if err != nil {
                log.Printf("ChatService: Error marshalling offline message %d for user %d: %v", msg.ID, userID, err)
                continue
            }

            // kirim pesan ke client
            select {
            // jika channel send client tidak penuh, kirim pesan
        case client.Send <- payloadBytes:
            log.Printf("ChatService: Offline message %d sent to user %d", msg.ID, userID)
            // jika channel send client penuh, log pesan dan lewati
            default:
                log.Printf("ChatService: Client %d's send channel is full, skipping message %d", userID, msg.ID)
            }
        }
    }
```

```
    }
    return nil
}

func (s *ChatServiceImpl) HandleIncomingMessage(ctx context.Context, senderID int, recipientID int, content string) error {
    // Validasi input
    if content == "" {
        return errors.New("message content cannot be empty")
    }
    if recipientID <= 0 {
        return errors.New("invalid recipient ID")
    }
    if senderID == recipientID {
        return errors.New("sender and recipient cannot be the same")
    }

    // step 1: buat pesan baru dalam bentuk entity.Message
    msg := entity.NewMessage(senderID, recipientID, content)

    // step 2: simpan pesan ke database
    if err := s.messageRepo.SaveMessage(ctx, msg); err != nil {
        log.Printf("ChatService: Error saving message from %d to %d: %v", senderID, recipientID, err)
        return fmt.Errorf("failed to save message: %w", err)
    }
    log.Printf("ChatService: Message from %d to %d saved successfully", senderID, recipientID)

    // step 3: kirimkan pesan ke penerima melalui WebSocket Hub
    s.hub.RoutePrivateMessage(msg)
    return nil
}

func (s *ChatServiceImpl) FetchConversationHistory(ctx context.Context, senderID, recipientID, limit, offset int) ([]*response.ChatM
    if limit <= 0 || offset < 0 {
        limit = 20 // default limit
        offset = 0 // default offset
    }

    // step 1: ambil history pesan dari repository
    messages, err := s.messageRepo.GetMessagesForConversation(ctx, senderID, recipientID, limit, offset)
    if err != nil {
        log.Printf("ChatService: Error fetching conversation history between %d and %d: %v", senderID, recipientID, err)
        return nil, fmt.Errorf("failed to fetch conversation history: %w", err)
    }

    // step 2: ubah pesan ke format response.ChatMessage dan scan setiap pesan
    var chatMessages []*response.ChatMessage
    for _, msg := range messages {
        chatMessages = append(chatMessages, &response.ChatMessage{
            ID: msg.ID,
            SenderID: msg.SenderID,
            RecipientID: msg.RecipientID,
            Content: msg.Content,
            Timestamp: msg.Timestamp,
            Status: msg.Status,
        })
    }

    log.Printf("ChatService: Fetched %d messages for conversation between %d and %d", len(chatMessages), senderID, recipientID)
    return chatMessages, nil
}

func (s *ChatServiceImpl) MarkMessageAsRead(ctx context.Context, messageID, userID int) error {
    if messageID <= 0 || userID <= 0 {
        return errors.New("invalid message ID or user ID")
    }
```

```
    // step 1: update status pesan ke 'read' di repository
    err := s.messageRepo.UpdateMessageStatus(ctx, messageID, entity.StatusRead)
    if err != nil {
        log.Printf("ChatService: Error marking message %d as read for user %d: %v", messageID, userID, err)
        return fmt.Errorf("failed to mark message as read: %w", err)
    }

    log.Printf("ChatService: Message %d marked as read for user %d", messageID, userID)
    return nil
}
```

service/websocket_hub.go

```
package service

/*
    WebScoket Hub Service:
    - berfungsi sebagai PENGHUBUNG UTAMA (pusat koordinasi = hub) antara client yang terhubung dengan server secara real-time
    - memiliki use-case sebagai berikut:
        1. Run(): menjalankan hub untuk menangani koneksi WebSocket, menerima pesan dari client, dan mengirim pesan ke client.
        2. RegisterClient(client *Client): mendaftarkan client baru yang terhubung ke hub saat koneksi WebSocket dibuka.
        3. UnregisterClient(client *Client): menghapus client yang terputus dari hub saat koneksi WebSocket ditutup.
        4. RoutePrivateMessage(message *entity.Message):
            - meneruskan pesan pribadi dari satu client ke client lain yang dituju.
            - misal: client A kirim pesan ke client B -> hub akan menerima pesan tersebut dan mengirimkannya ke client B.
*/

import (
    "context"
    "encoding/json"
    "log"
    "mood-bridge-v2/server/internal/entity"
    "mood-bridge-v2/server/internal/model/response"
    "mood-bridge-v2/server/internal/repository"
    "sync"
)

type Hub interface {
    Run()
    RegisterClient(client *Client)
    UnregisterClient(client *Client)
    RoutePrivateMessage(message *entity.Message)
}

type HubImpl struct { // berfungsi untuk menyimpan daftar client yang aktif (yang terhubung via WebSocket) dan menyediakan cara untu
    // menyimpan daftar clients yang terhubung dengan userID sebagai key
    clients map[int]*Client
    // mengamankan akses clients agar thread-safe saat ada banyak koneksi WebSocket (intinya saat ada banyak client yang terhubung k
    clientsMutex sync.RWMutex
    // menyimpan pesan ke database, jadi tidak hanya dikirim tapi juga disimpan untuk riwayat chat
    messageRepo repository.ChatRepository
}

func NewConcreteHub(msgRepo repository.ChatRepository) Hub {
    return &HubImpl{
        clients: make(map[int]*Client),
        messageRepo: msgRepo,
    }
}

func (h *HubImpl) Run() {
    log.Println("WebSocket Hub is running...")
}
```

```
func (h *HubImpl) RegisterClient(client *Client) {
    // step 1: kunci thread (mutex) untuk mengamankan akses ke map clients
    h.clientsMutex.Lock()

    // step 2: unlock mutex jika sudah selesai
    defer h.clientsMutex.Unlock()

    // step 3: tambahkan client ke map clients
    log.Printf("Registering client: %d", client.UserID)
    h.clients[client.UserID] = client

    // step 4: jalankan writePump dan readPump untuk client yang baru terdaftar
    go client.WritePump()
    go client.ReadPump()
}


func (h *HubImpl) UnregisterClient(client *Client) {
    // step 1: kunci thread (mutex) untuk mengamankan akses ke map clients
    h.clientsMutex.Lock()

    // step 2: unlock mutex jika sudah selesai
    defer h.clientsMutex.Unlock()

    // step 3: hapus client dari map clients
    if _, ok := h.clients[client.UserID]; ok {
        log.Printf("Unregistering client: %d", client.UserID)
        delete(h.clients, client.UserID)
        close(client.Send)
    }
}

func (h *HubImpl) RoutePrivateMessage(message *entity.Message) {
    // step 1: kunci thread (mutex) untuk mengamankan akses ke map clients
    h.clientsMutex.RLock()

    // step 2: tentukan apakah penerima client ada di map clients
    recipientClient, ok := h.clients[message.RecipientID]

    // step 3: unlock mutex jika sudah selesai
    defer h.clientsMutex.RUnlock()

    // step 4: jika penerima client ada, kirim pesan ke penerima
    log.Printf("Routing private message from %d to %d: %s", message.SenderID, message.RecipientID, message.Content)
    if ok {
        // jika recipientClient online, coba kirim pesan ke recipientClient dalam bentuk json (step ini pertama ubah dulu ke bentuk
        wsMsg := response.WebSocketMessage{
            Type: "new_private_message",
            Payload: response.ChatMessage{
                ID: message.ID,
                SenderID: message.SenderID,
                RecipientID: message.RecipientID,
                Content: message.Content,
                Timestamp: message.Timestamp,
                Status: message.Status, // status pesan ubah jadi "sent"
            },
        }
        payloadBytes, err := json.Marshal(wsMsg)
        if err != nil {
            log.Printf("Error marshalling message for recipient %d: %v", message.RecipientID, err)
            return
        }

        // step 5: kirim pesan ke recipientClient
        select {
```

```
        case recipientClient.Send <- payloadBytes:
            log.Printf("Message sent to recipient %d: %s", message.RecipientID, message.Content)
            // jika pesan berhasil dikirim, update status pesan di database
            if h.messageRepo != nil {
                go func() {
                    err := h.messageRepo.UpdateMessageStatus(context.Background(), message.ID, entity.StatusDelivered)
                    if err != nil {
                        log.Printf("Error updating message status for message ID %d: %v", message.ID, err)
                    } else {
                        log.Printf("Message status updated to 'delivered' for message ID %d", message.ID)
                    }
                }()
            }
        default:
            log.Printf("Hub: Send channel full for recipient %d. Message ID: %d", message.RecipientID, message.ID) // artinya pesan
        }
    } else {
        // step 6: jika client penerima tidak ada (offline), simpan pesan ke database dan akan diambil saat client reconnect
        log.Printf("Recipient client %d is offline. Message ID: %d stored in DB", message.RecipientID, message.ID)
    }
}
```

service/websocket_client.go

```
package service

/*
    WebSocket Client Service:
    - WebSocket adalah protokol full-duplex (dua arah) yang memungkinkan komunikasi real-time antara client dan server.
    - Setiap client yang terhubung lewat WebSocket membutuhkan sebuah struct Client yang menyimpan informasi tentang koneksi, user I
    - Client memiliki dua goroutine utama: ReadPump untuk membaca pesan dari client dan WritePump untuk mengirim pesan ke client.
    - ReadPump menangani pesan masuk dari client, memprosesnya, dan meneruskan pesan tersebut ke ChatService untuk penanganan lebih
    - WritePump menangani pengiriman pesan keluar ke client, termasuk pesan offline yang mungkin diterima saat client tidak terhubun
    - Client juga menangani ping/pong untuk menjaga koneksi tetap hidup dan mendeteksi jika client terputus.
    - Client ini diibaratkan sebagai jembatan antara WebSocket dan ChatService
*/

import (
    "encoding/json"
    "log"
    "mood-bridge-v2/server/internal/model/request"
    "mood-bridge-v2/server/internal/model/response"
    "time"

    "github.com/gorilla/websocket"
)

const (
    writeWait = 10 * time.Second // Timeout saat kirim pesan
    pongWait = 60 * time.Second // Timeout jika client tidak merespon ping
    pingPeriod = (pongWait * 9) / 10 // Waktu kirim ping ke client
    maxMessageSize = 1024 * 4 // Maksimum ukuran pesan yang diterima dari client
)

var newline = '\n'

type Client struct {
    UserID int // ID unik untuk mengidentifikasi client
    Hub Hub // Hub yang mengelola koneksi client
    Conn *websocket.Conn // Koneksi WebSocket untuk client yang aktif
    Send chan []byte // Channel untuk mengirim pesan ke client
    ChatService ChatService // Service yang menangani logika chat, seperti mengirim pesan, mengambil riwayat chat, dll.
}
```

```
func NewClient(userID int, hub Hub, conn *websocket.Conn, chatService ChatService) *Client {
    return &Client{
        UserID: userID,
        Hub: hub,
        Conn: conn,
        Send: make(chan []byte, 256),
        ChatService: chatService,
    }
}

func (c *Client) ReadPump() {
    // step 1: ketika koneksi terputus, unregister client dari Hub dan tutup koneksi
    defer func() {
        c.Hub.UnregisterClient(c)
        c.Conn.Close()
        log.Printf("Client %d disconnected, readPump closed", c.UserID)
    }()

    // step 2: atur batasan ukuran pesan yang diterima dari client dan atur timeout untuk membaca pesan
    c.Conn.SetReadLimit(maxMessageSize)
    _ = c.Conn.SetReadDeadline(time.Now().Add(pongWait))

    // step 3: atur handler dengan memperbarui deadline setiap kali menerima pong dari client
    c.Conn.SetPongHandler(func(string) error {
        _ = c.Conn.SetReadDeadline(time.Now().Add(pongWait));
        return nil
    })

    // step 4: baca pesan dari client dalam loop
    for {
        messageType, messageBytes, err := c.Conn.ReadMessage()
        if err != nil { // jika terjadi error, keluar dari loop
            if websocket.IsUnexpectedCloseError(err, websocket.CloseGoingAway, websocket.CloseAbnormalClosure) {
                log.Printf("error reading message for client %d: %v", c.UserID, err)
            }
            break
        }

        // step 5: jika pesan yang diterima adalah teks, proses pesan tersebut
        if messageType == websocket.TextMessage {
            var msgPayload request.PrivateMessagePayload
            if err := json.Unmarshal(messageBytes, &msgPayload); err != nil {
                log.Printf("Error unmarshalling message from client %d: %v. Message: %s", c.UserID, err, string(messageBytes))
                errorMsg := response.WebSocketMessage{
                    Type: "error",
                    Payload: response.ErrorMessage{Code: "invalid_message", Message: "Invalid message format"},
                }
                errorBytes, _ := json.Marshal(errorMsg)
                select {
                case c.Send <- errorBytes:
                default:
                    log.Printf("Error sending error message to client %d: send channel is full", c.UserID)
                }
                continue
            }

            // step 6: jika pesan valid, kirim pesan ke ChatService untuk diproses
            err := c.ChatService.HandleIncomingMessage(nil, c.UserID, msgPayload.RecipientID, msgPayload.Content)
            if err != nil {
                log.Printf("Error from ChatService.HandleIncomingMessage for client %d: %v", c.UserID, err)
                errMsgPayload := response.WebSocketMessage{
                    Type: "message_send_failed",
                    Payload: response.ErrorMessage{
                        Code: "send_error",
                        Message: err.Error(),
```

```
                    },
                }
                errMsgBytes, _ := json.Marshal(errMsgPayload)
                select {
                    case c.Send <- errMsgBytes:
                    default:
                    log.Printf("Error sending error message to client %d: send channel is full", c.UserID)
                }
            }
        }
    }
}

func (c *Client) WritePump() {
    // step 1: define timer untuk mengirim ping ke client secara berkala
    ticker := time.NewTicker(pingPeriod)
    defer func(){
        ticker.Stop()
        c.Conn.Close()
        log.Printf("Client %d disconnected, writePump closed", c.UserID)
    }()
    for {
        select {
        case message, ok := <-c.Send:
            // step 2: set timeout untuk menulis pesan ke client
            _ = c.Conn.SetWriteDeadline(time.Now().Add(writeWait))

            // step 3: jika channel Send sudah ditutup, kirim pesan close ke client dan keluar dari loop
            if !ok {
                _ = c.Conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            // step 4: jika pesan yang diterima adalah teks, kirim pesan ke client
            w, err := c.Conn.NextWriter(websocket.TextMessage)
            if err != nil {
                log.Printf("Error getting next writer for client %d: %v", c.UserID, err)
                return
            }
            _, _ = w.Write(message)

            // step 5: jika ada pesan lain yang ada di channel Send, kirim pesan tersebut ke client (biasanya pesan offline)
            n := len(c.Send)
            for i := 0; i < n; i++ {
                _, _ = w.Write([]byte{byte(newline)})
                _, _ = w.Write(<-c.Send)
            }
            if err := w.Close(); err != nil {
                return
            }

            // step 6: kirim ping tiap interval untuk menjaga koneksi tetap hidup
        case <- ticker.C:
            _ = c.Conn.SetWriteDeadline(time.Now().Add(writeWait))
            if err := c.Conn.WriteMessage(websocket.PingMessage, nil); err != nil {
                log.Printf("Error sending ping to client %d: %v", c.UserID, err)
                return
            }
        }
    }
}
```

handler/chat_handler.go

```
package handler

import (
    "log"
    "mood-bridge-v2/server/internal/service"
    "net/http"
    "strconv"

    "github.com/gin-gonic/gin"
    "github.com/gorilla/websocket"
)

type ChatHandler interface {
    HandleWebSocketConnection(c *gin.Context)
    HandleFetchChatHistory(c *gin.Context)
    HandleMarkMessageAsRead(c *gin.Context)
}

type ChatHandlerImpl struct {
    chatService service.ChatService
    upgrader websocket.Upgrader
}

func NewChatHandler(chatService service.ChatService) ChatHandler {
    return &ChatHandlerImpl{
        chatService: chatService,
        upgrader: websocket.Upgrader{
            ReadBufferSize: 1024,
            WriteBufferSize: 1024,
            CheckOrigin: func(r *http.Request) bool {
                origin := r.Header.Get("Origin")
                log.Printf("Handler: WebSocket connection request from origin: %s", origin)
                return true // Allow all origins for simplicity, adjust as needed
            },
        },
    }
}

func (h *ChatHandlerImpl) HandleWebSocketConnection(c *gin.Context) {
    // step 1: ambil userID dari auth middleware
    userIDInterface := c.Request.Context().Value("userID")
    if userIDInterface == nil {
        log.Println("Handler: User ID not found in context")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
        })
        return
    }

    // step 2: ubah ke int
    userID, ok := userIDInterface.(int)
    if !ok {
        log.Println("Handler: User ID is not an integer")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
        })
        return
    }

    log.Printf("Handler: Attempting WebSocket upgrade for UserID: %d", userID)// buat test doang

    // step 2: upgrade koneksi ke WebSocket
    conn, err := h.upgrader.Upgrade(c.Writer, c.Request, nil)
```

```
    if err != nil {
        log.Printf("Handler: Failed to upgrade connection: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Failed to upgrade connection",
        })
        return
    }

    log.Printf("Handler: WebSocket connection upgraded for UserID: %d", userID)

    // step 3: panggil service untuk menangani koneksi WebSocket
    err = h.chatService.HandleNewConnection(c.Request.Context(), userID, conn)
    if err != nil {
        log.Printf("Handler: Error handling new connection: %v", err)
        _ = conn.Close() // close the connection if there's an error
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Failed to handle new connection",
        })
        return
    }
}

func (h *ChatHandlerImpl) HandleFetchChatHistory(c *gin.Context) {
    // step 1: ambil userID dari auth middleware
    userIDInterface := c.Request.Context().Value("userID")
    if userIDInterface == nil {
        log.Println("Handler: User ID not found in context")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
        })
        return
    }

    // step 2: ubah ke int
    userID, ok := userIDInterface.(int)
    if !ok {
        log.Println("Handler: User ID is not an integer")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
        })
        return
    }

    // step 3: ambil recipientID dari query parameter
    recipientIDstr := c.Query("with_user_id")
    if recipientIDstr == "" {
        log.Println("Handler: Recipient ID not provided")
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Recipient ID is required",
        })
        return
    }

    recipientID, err := strconv.Atoi(recipientIDstr)
    if err != nil || recipientID <= 0 {
        log.Printf("Handler: Invalid recipient ID: %s", recipientIDstr)
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Invalid recipient ID",
        })
```

```
        return
    }

    // step 4: ambil limit dan offset dari query parameter
    limitStr := c.DefaultQuery("limit", "20")
    limit, err := strconv.Atoi(limitStr)
    if err != nil || limit <= 0 {
        log.Printf("Handler: Invalid limit: %s", limitStr)
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Invalid limit",
        })
        return
    }

    offsetStr := c.DefaultQuery("offset", "0")
    offset, err := strconv.Atoi(offsetStr)
    if err != nil || offset < 0 {
        log.Printf("Handler: Invalid offset: %s", offsetStr)
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Invalid offset",
        })
        return
    }

    // step 5: panggil service untuk mengambil chat history
    messages, err := h.chatService.FetchConversationHistory(c.Request.Context(), userID, recipientID, limit, offset)
    if err != nil {
        log.Printf("Handler: Error fetching chat history: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Failed to fetch chat history",
        })
        return
    }

    log.Printf("Handler: Successfully fetched chat history for UserID: %d with RecipientID: %d", userID, recipientID)
    // step 6: kirim response dengan chat history
    c.JSON(http.StatusOK, gin.H{
        "code":    http.StatusOK,
        "message": "Chat history fetched successfully",
        "data":    messages,
    })
    log.Printf("Handler: Chat history response sent for UserID: %d with RecipientID: %d", userID, recipientID)
}

func (h *ChatHandlerImpl) HandleMarkMessageAsRead(c *gin.Context) {
    // step 1: ambil userID dari auth middleware
    userIDInterface := c.Request.Context().Value("userID")
    if userIDInterface == nil {
        log.Println("Handler: User ID not found in context")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
        })
        return
    }

    // step 2: ubah ke int
    userID, ok := userIDInterface.(int)
    if !ok {
        log.Println("Handler: User ID is not an integer")
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Internal Server Error",
```

```
        })
        return
    }

    // step 3: ambil messageID dari query parameter
    messageIDStr :=  c.Param("message_id")
    if messageIDStr == "" {
        log.Println("Handler: Message ID not provided")
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Message ID is required",
        })
        return
    }

    // step 4: ubah messageID ke int
    messageID, err := strconv.Atoi(messageIDStr)
    if err != nil || messageID <= 0 {
        log.Printf("Handler: Invalid message ID: %s", messageIDStr)
        c.JSON(http.StatusBadRequest, gin.H{
            "code":    http.StatusBadRequest,
            "message": "Invalid message ID",
        })
        return
    }

    // step 5: panggil service untuk menandai pesan sebagai dibaca
    err = h.chatService.MarkMessageAsRead(c.Request.Context(), userID, messageID)
    if err != nil {
        log.Printf("Handler: Error marking message as read: %v", err)
        c.JSON(http.StatusInternalServerError, gin.H{
            "code":    http.StatusInternalServerError,
            "message": "Failed to mark message as read",
        })
        return
    }

    log.Printf("Handler: Successfully marked message as read for UserID: %d, MessageID: %d", userID, messageID)
    c.JSON(http.StatusOK, gin.H{
        "code":    http.StatusOK,
        "message": "Message marked as read successfully",
        "data":    nil,
    })
}
```