LISTING PROGRAM

Database.go

```
package app

import (
	"log"
	"mikti-depublic/model/domain"
	"os"

	"gorm.io/driver/postgres"
	"gorm.io/gorm"
)

var DB *gorm.DB

func DBConnection() {
	var err error
	dsn := os.Getenv("DB_POSTGRES_URL")
	DB, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})

	if err != nil {
		log.Fatal("Failed to connect to database")
	}

	err = DB.AutoMigrate(&domain.User{}, &domain.Admin{},
&domain.Event{}, &domain.Transaction{})
	if err != nil {
		log.Fatalf("Failed to migrate the database schema:
%v", err)
	}
}

func GetDB() *gorm.DB {
	return DB
}
```

Kode di atas merupakan implementasi fungsi koneksi database menggunakan GORM dengan PostgreSQL dalam proyek Golang. Variabel global DB digunakan untuk menyimpan instance koneksi database yang dapat diakses di seluruh aplikasi. Fungsi DBConnection() bertanggung jawab untuk menginisialisasi koneksi ke database PostgreSQL dengan mengambil Data Source Name (DSN) dari variabel lingkungan (os.Getenv("DB_POSTGRES_URL")). Jika koneksi gagal, aplikasi akan menampilkan pesan error dan langsung berhenti (log.Fatal). Setelah koneksi berhasil, proses migrasi otomatis dilakukan dengan DB.AutoMigrate(), yang memastikan bahwa struktur tabel untuk entitas User, Admin, Event, dan Transaction sesuai dengan definisi dalam paket domain. Jika migrasi gagal, program juga akan menampilkan pesan error dan menghentikan eksekusi. Fungsi GetDB() disediakan agar instance database dapat diakses dari bagian lain dalam aplikasi, memastikan penggunaan database yang terpusat dan terkelola dengan baik.

History_controller.go

```
package controller

import (
	"mikti-depublic/service"

	"github.com/labstack/echo/v4"
)

type HistoryController interface {
	GetHistory(c echo.Context) error
	GetHistoryByID(c echo.Context) error
	GetHistoryByStatus(c echo.Context) error
}

func NewHistoryController(historyService service.HistoryService)
HistoryController {
	return &HistoryControllerImpl{historyService:
historyService}
```

```
}
```

Kode di atas merupakan implementasi controller dalam arsitektur MVC (Model-View-Controller) menggunakan Echo framework dalam Golang. HistoryController adalah sebuah interface yang mendefinisikan tiga metode utama: GetHistory(c echo.Context) error, GetHistoryByID(c echo.Context) error, dan GetHistoryByStatus(c echo.Context) error. Ketiga metode ini berfungsi untuk mengambil data histori secara keseluruhan, berdasarkan ID, dan berdasarkan status. Fungsi NewHistoryController(historyService service.HistoryService) digunakan sebagai constructor untuk menginisialisasi dan mengembalikan implementasi dari HistoryController. Namun, dalam kode yang diberikan, implementasi HistoryControllerImpl belum ditampilkan, sehingga kode ini masih belum lengkap. Untuk menyempurnakan fungsionalitasnya, perlu ada struktur HistoryControllerImpl yang mengimplementasikan interface tersebut dan memanfaatkan dependency injection dengan menerima HistoryService, yang bertanggung jawab untuk mengelola logika bisnis terkait histori.

History_controller_impl.go

```
package controller

import (
	"mikti-depublic/helper"
	"mikti-depublic/model"
	"mikti-depublic/model/domain"
	"mikti-depublic/service"
	"net/http"
	"strings"

	"github.com/labstack/echo/v4"
)

type HistoryControllerImpl struct {
	historyService service.HistoryService
```

```
}

func NewHistoryControllerImpl(historyService
service.HistoryService) *HistoryControllerImpl {
	return &HistoryControllerImpl{historyService:
historyService}
}

func (controller *HistoryControllerImpl) GetHistory(c
echo.Context) error {
	userPayload, ok :=
c.Get("claims").(helper.JwtCustomClaims)
	if !ok {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError, "Unable
to retrieve user claims", nil))
	}

	var histories []domain.Transaction
	var err error

	if strings.HasPrefix(userPayload.ID, "ADMIN") {
		histories, err =
controller.historyService.GetHistories()
		if err != nil {
			return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError,
err.Error(), nil))
		}
	} else {
		histories, err =
controller.historyService.GetHistoriesForUser(userPayload.ID)
		if err != nil {
			return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError,
err.Error(), nil))
		}
```

```
	}

	return c.JSON(http.StatusOK,
model.ResponseToClient(http.StatusOK, "Success Get Histories",
histories))
}

func (controller *HistoryControllerImpl) GetHistoryByID(c
echo.Context) error {
	id := c.Param("id")
	userPayload, ok :=
c.Get("claims").(helper.JwtCustomClaims)
	if !ok {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError, "Unable
to retrieve user claims", nil))
	}

	history, err :=
controller.historyService.GetHistoryByID(id)
	if err != nil {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError,
err.Error(), nil))
	}

	if strings.HasPrefix(userPayload.ID, "ADMIN") ||
userPayload.ID == id {
		if history == nil {
			return c.JSON(http.StatusNotFound,
model.ResponseToClient(http.StatusNotFound, "History not found",
nil))
		}
	} else {
		return c.JSON(http.StatusForbidden,
model.ResponseToClient(http.StatusForbidden, "Access denied",
nil))
```

```
	}

	return c.JSON(http.StatusOK,
model.ResponseToClient(http.StatusOK, "Success Get History",
history))
}

func (controller *HistoryControllerImpl) GetHistoriesForUser(c
echo.Context) error {
	id := c.Param("id")

	userPayload, ok :=
c.Get("claims").(helper.JwtCustomClaims)
	if !ok {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError, "Unable
to retrieve user claims", nil))
	}

	if !strings.HasPrefix(userPayload.ID, "ADMIN") &&
userPayload.ID != id {
		return c.JSON(http.StatusForbidden,
model.ResponseToClient(http.StatusForbidden, "Access denied",
nil))
	}

	histories, err :=
controller.historyService.GetHistoriesForUser(id)
	if err != nil {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError,
err.Error(), nil))
	}

	return c.JSON(http.StatusOK,
model.ResponseToClient(http.StatusOK, "Success Get Histories",
histories))
```

```
}


func (controller *HistoryControllerImpl) GetHistoryByStatus(c
echo.Context) error {
	status := c.Param("status")
	userPayload, ok :=
c.Get("claims").(helper.JwtCustomClaims)
	if !ok {
		return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError, "Unable
to retrieve user claims", nil))
	}

	var histories []domain.Transaction
	var err error

	if strings.HasPrefix(userPayload.ID, "ADMIN") {
		histories, err =
controller.historyService.GetHistoryByStatus(status)
		if err != nil {
			return c.JSON(http.StatusInternalServerError,
model.ResponseToClient(http.StatusInternalServerError,
err.Error(), nil))
		}
	} else {
		return c.JSON(http.StatusForbidden,
model.ResponseToClient(http.StatusForbidden, "Access denied",
nil))
	}

	return c.JSON(http.StatusOK,
model.ResponseToClient(http.StatusOK, "Success Get History By
Status", histories))
}
```

Kode di atas merupakan implementasi dari HistoryControllerImpl, sebuah controller dalam Echo framework yang menangani permintaan terkait histori transaksi. Controller ini berkomunikasi dengan HistoryService untuk mengambil data yang diperlukan. Fungsi NewHistoryControllerImpl digunakan untuk menginisialisasi HistoryControllerImpl dengan dependensi historyService.

Fungsi GetHistory mengambil semua histori transaksi, dengan perbedaan akses berdasarkan peran pengguna. Jika pengguna memiliki ID dengan prefix "ADMIN", mereka dapat melihat seluruh histori transaksi, sedangkan pengguna biasa hanya bisa melihat histori mereka sendiri dengan memanggil GetHistoriesForUser().

Fungsi GetHistoryByID mengambil histori transaksi berdasarkan ID transaksi. Administrator dapat mengakses semua histori, sedangkan pengguna biasa hanya dapat mengakses histori milik mereka sendiri. Jika histori tidak ditemukan, akan dikembalikan respons 404 Not Found, dan jika pengguna tidak memiliki akses, akan dikembalikan 403 Forbidden.

Fungsi GetHistoriesForUser mengambil seluruh histori transaksi berdasarkan ID pengguna. Sama seperti fungsi sebelumnya, hanya administrator atau pemilik histori yang bisa mengakses data ini. Jika pengguna lain mencoba mengakses data orang lain tanpa hak akses, permintaan akan ditolak dengan 403 Forbidden.

Fungsi GetHistoryByStatus mengambil daftar histori berdasarkan status transaksi. Hanya pengguna dengan peran ADMIN yang dapat mengakses data ini, sedangkan pengguna biasa akan mendapatkan respons 403 Forbidden jika mencoba mengaksesnya.

Setiap fungsi menggunakan middleware JWT untuk mendapatkan informasi pengguna melalui c.Get("claims"). Jika klaim tidak bisa diambil, maka akan dikembalikan 500 Internal Server Error. Untuk setiap respons, kode menggunakan model.ResponseToClient(), yang kemungkinan merupakan helper untuk memformat respons JSON secara konsisten dalam aplikasi.

Token.go

```
package helper

import (
	"errors"
	"os"

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

type TokenUseCase interface {
	GenerateAccessToken(claims JwtCustomClaims) (string,
error)
	DecodeTokenPayload(token string) (JwtCustomClaims, error)
}

type TokenUseCaseImpl struct{}

type JwtCustomClaims struct {
	ID    string `json:"user_id"`
	Name  string `json:"name"`
	Email string `json:"email"`
	jwt.RegisteredClaims
}

func NewTokenUseCase() *TokenUseCaseImpl {
	return &TokenUseCaseImpl{}
}

func (t *TokenUseCaseImpl) GenerateAccessToken(claims
JwtCustomClaims) (string, error) {
	plainToken := jwt.NewWithClaims(jwt.SigningMethodHS256,
claims)

	encodedToken, err :=
plainToken.SignedString([]byte(os.Getenv("SECRET_KEY")))
	if err != nil {
```

```
            return "", err
      }

      return encodedToken, nil

}

func (t *TokenUseCaseImpl) DecodeTokenPayload(token string)
(JwtCustomClaims, error) {
      payload, err := jwt.ParseWithClaims(token,
&JwtCustomClaims{}, func(jwtToken *jwt.Token) (interface{},
error) {
            return []byte(os.Getenv("SECRET_KEY")), nil
      })

      if err != nil {
            return JwtCustomClaims{}, err
      }

      claims, ok := payload.Claims.(*JwtCustomClaims)
      if ok && payload.Valid {
            return *claims, nil
      }
      return JwtCustomClaims{}, errors.New("failed when decode
payload")
}
```

Kode di atas merupakan implementasi dari TokenUseCase, sebuah helper dalam Golang yang bertanggung jawab untuk menangani pembuatan dan validasi JWT (JSON Web Token) menggunakan paket github.com/golang-jwt/jwt/v5. Interface TokenUseCase mendefinisikan dua metode utama: GenerateAccessToken(claims JwtCustomClaims) untuk menghasilkan token akses dan DecodeTokenPayload(token string) untuk mendekode payload dari token JWT.

Struktur JwtCustomClaims digunakan untuk menyimpan klaim khusus dalam token JWT, yang mencakup ID pengguna, nama, email, serta klaim terdaftar

dari jwt.RegisteredClaims. Struktur TokenUseCaseImpl merupakan implementasi konkret dari TokenUseCase dan digunakan untuk mengelola token.

Fungsi GenerateAccessToken bertugas membuat token JWT dengan algoritma HS256, menggunakan SECRET_KEY yang diambil dari variabel lingkungan (os.Getenv("SECRET_KEY")). Jika terjadi kesalahan saat proses penandatanganan token (SignedString), maka fungsi akan mengembalikan error.

Fungsi DecodeTokenPayload bertanggung jawab untuk memverifikasi dan mengekstrak payload dari token JWT. Fungsi ini menggunakan jwt.ParseWithClaims untuk mendekode token dengan JwtCustomClaims sebagai struktur klaimnya. Jika token valid dan berhasil di-dekode, fungsi akan mengembalikan klaim dalam bentuk struct JwtCustomClaims. Namun, jika token tidak valid atau gagal diproses, maka akan dikembalikan error "failed when decode payload".

Secara keseluruhan, kode ini memberikan solusi yang aman untuk autentikasi berbasis JWT dalam aplikasi Golang dengan memastikan bahwa token hanya bisa dibuat dan divalidasi menggunakan kunci rahasia yang telah ditentukan.

Jwt_token_validator.go

```
package middleware

import (
	"mikti-depublic/helper"
	"net/http"
	"strings"

	"github.com/labstack/echo/v4"
)

func JwtTokenValidator(next echo.HandlerFunc) echo.HandlerFunc {
	return func(c echo.Context) error {
		authHeader :=
c.Request().Header.Get("Authorization")
```

```
            if authHeader == "" {
                  return c.JSON(http.StatusUnauthorized,
map[string]string{
                        "message": "Missing or invalid token",
                  })
            }

            const bearerPrefix = "Bearer "
            if len(authHeader) < len(bearerPrefix) ||
authHeader[:len(bearerPrefix)] != bearerPrefix {
                  return c.JSON(http.StatusUnauthorized,
map[string]string{
                        "message": "Missing or invalid token",
                  })
            }
            token := authHeader[len(bearerPrefix):]

            tokenUseCase := helper.NewTokenUseCase()
            claims, err :=
tokenUseCase.DecodeTokenPayload(token)
            if err != nil {
                  return c.JSON(http.StatusUnauthorized,
map[string]string{
                        "message": "Invalid or expired token",
                  })
            }

            c.Set("claims", claims)

            userID := claims.ID

            requestedPath := c.Path()

            accessControl := map[string]func(userID, path
string) bool{
                  "/event/createEvent": func(id, _ string) bool
{ return strings.HasPrefix(id, "ADMIN") },
```

```
                  "/event/:id":         func(id, _ string) bool
{ return strings.HasPrefix(id, "ADMIN") },
                  "/history":           func(id, _ string) bool
{ return strings.HasPrefix(id, "ADMIN") },
                  "/history/:id":       func(id, path string)
bool { return strings.HasPrefix(id, "ADMIN") || id ==
c.Param("id") },
                  "/history/user/:id":  func(id, path string)
bool { return strings.HasPrefix(id, "ADMIN") || id ==
c.Param("id") },
                  "/history/status/:status": func(id, _ string)
bool { return strings.HasPrefix(id, "ADMIN") },
            }

            for path, checkFunc := range accessControl {
                  if strings.HasPrefix(requestedPath, path) {
                        if !checkFunc(userID, requestedPath) {
                              return
c.JSON(http.StatusForbidden, map[string]string{
                                    "message": "Access denied",
                              })
                        }
                        break
                  }
            }

            return next(c)
      }
}
```

Kode di atas adalah middleware JwtTokenValidator yang digunakan dalam Echo framework untuk melakukan validasi JWT (JSON Web Token) pada setiap permintaan yang masuk. Middleware ini berfungsi untuk memastikan bahwa hanya pengguna yang memiliki token yang sah dan sesuai dengan aturan akses yang ditentukan yang dapat mengakses endpoint tertentu.

Middleware ini pertama-tama mengambil header Authorization dari permintaan dan memeriksa apakah token tersedia serta memiliki format yang benar (Bearer). Jika tidak, permintaan akan langsung dikembalikan dengan status 401 Unauthorized. Jika token tersedia, middleware akan menggunakan helper.NewTokenUseCase().DecodeTokenPayload(token) untuk mendekode token dan mendapatkan klaim pengguna, seperti ID pengguna. Jika proses dekode gagal, berarti token tidak valid atau sudah kedaluwarsa, sehingga permintaan juga akan ditolak dengan status 401 Unauthorized.

Setelah token berhasil divalidasi, klaim pengguna disimpan di context (c.Set("claims", claims")), sehingga bisa digunakan oleh handler lain dalam aplikasi. Middleware kemudian melakukan kontrol akses berdasarkan ID pengguna dan endpoint yang diakses. Mapping aturan akses disimpan dalam accessControl, yang merupakan peta dari endpoint ke fungsi pengecekan hak akses. Misalnya, hanya pengguna dengan ID yang dimulai dengan "ADMIN" yang dapat mengakses endpoint /event/createEvent dan /history. Sementara itu, pengguna biasa hanya bisa melihat histori transaksi mereka sendiri dengan memeriksa apakah ID pengguna yang diautentikasi cocok dengan ID yang ada di URL parameter.

Jika pengguna tidak memiliki izin untuk mengakses suatu endpoint, middleware akan mengembalikan status 403 Forbidden dengan pesan "Access denied". Jika semua pemeriksaan lolos, middleware akan meneruskan eksekusi ke handler berikutnya dengan next(c). Dengan cara ini, middleware memastikan keamanan endpoint dengan membatasi akses berdasarkan peran pengguna dan validitas token.

History_Repository.go

```
package repository


import (
	"mikti-depublic/model/domain"
)
```

```
type HistoryRepository interface {
	Create(transaction *domain.Transaction) error
	FindAll() ([]domain.Transaction, error)
	FindByID(id string) (*domain.Transaction, error)
	FindByStatus(status string) ([]domain.Transaction, error)
	FindByUserID(userID string) ([]domain.Transaction, error)
}

func (r *historyRepositoryImpl) FindByUserID(userID string)
([]domain.Transaction, error) {
	var transactions []domain.Transaction
	if err := r.db.Where("users_id = ?",
userID).Find(&transactions).Error; err != nil {
		return nil, err
	}
	return transactions, nil
}
```

Kode di atas merupakan implementasi dari HistoryRepository, sebuah interface dalam arsitektur repository pattern yang bertanggung jawab untuk mengelola operasi database terkait entitas Transaction dalam aplikasi Golang. Interface ini mendefinisikan lima metode utama: Create(transaction *domain.Transaction) error untuk menambahkan transaksi baru, FindAll() ([]domain.Transaction, error) untuk mengambil semua transaksi, FindByID(id string) (*domain.Transaction, error) untuk mencari transaksi berdasarkan ID, FindByStatus(status string) ([]domain.Transaction, error) untuk mencari transaksi berdasarkan status, dan FindByUserID(userID string) ([]domain.Transaction, error) untuk mendapatkan semua transaksi milik pengguna tertentu.

Fungsi FindByUserID merupakan implementasi dari salah satu metode dalam HistoryRepository. Fungsi ini menerima ID pengguna (userID) sebagai parameter dan melakukan pencarian transaksi di database menggunakan GORM ORM. Query r.db.Where("users_id = ?", userID).Find(&transactions).Error akan

mencari semua transaksi yang memiliki users_id sesuai dengan parameter yang diberikan. Jika query berhasil, fungsi akan mengembalikan slice berisi daftar transaksi. Namun, jika terjadi error dalam proses pencarian, fungsi akan mengembalikan nil beserta error yang terjadi.

Kode ini mendukung pemrosesan histori transaksi berdasarkan pengguna, yang dapat digunakan oleh service layer untuk menampilkan riwayat transaksi individu. Namun, dalam kode yang diberikan, tidak ada deklarasi struct historyRepositoryImpl yang berisi objek db, sehingga implementasi repository ini masih belum lengkap dan perlu ditambahkan agar bisa digunakan dalam aplikasi.

History_Repository_Impl.go

```
package repository

import (
	"mikti-depublic/model/domain"

	"gorm.io/gorm"
)

type historyRepositoryImpl struct {
	db *gorm.DB
}

func NewHistoryRepositoryImpl(db *gorm.DB) HistoryRepository {
	return &historyRepositoryImpl{db: db}
}

func (r *historyRepositoryImpl) Create(transaction
*domain.Transaction) error {
	return r.db.Create(transaction).Error
}

func (r *historyRepositoryImpl) FindAll() ([]domain.Transaction,
error) {
	var histories []domain.Transaction
```

```
	if err := r.db.Find(&histories).Error; err != nil {
		return nil, err
	}
	return histories, nil
}

func (r *historyRepositoryImpl) FindByID(id string)
(*domain.Transaction, error) {
	var histories domain.Transaction
	if err := r.db.First(&histories, "id = ?", id).Error; err
!= nil {
		return nil, err
	}
	return &histories, nil
}

func (r *historyRepositoryImpl) FindByStatus(status string)
([]domain.Transaction, error) {
	var histories []domain.Transaction
	if err := r.db.Where("status = ?",
status).Find(&histories).Error; err != nil {
		return nil, err
	}
	return histories, nil
}
```

Kode di atas merupakan implementasi dari repository pattern dalam aplikasi Golang menggunakan GORM sebagai ORM untuk mengelola transaksi dalam database. Struct historyRepositoryImpl didefinisikan dengan atribut db bertipe *gorm.DB, yang digunakan untuk melakukan operasi database. Kode ini juga menyediakan fungsi NewHistoryRepositoryImpl, sebuah constructor function yang mengembalikan instance dari historyRepositoryImpl dengan menyertakan objek db sebagai dependensi.

Beberapa metode utama dalam repository ini adalah Create, yang bertugas untuk menambahkan data transaksi baru ke dalam database menggunakan

db.Create(transaction). Metode FindAll mengambil semua data transaksi yang tersedia dengan db.Find(&histories), sementara FindByID mencari transaksi berdasarkan ID tertentu dengan db.First(&histories, "id = ?", id). Untuk mengambil transaksi berdasarkan status, digunakan metode FindByStatus, yang memanfaatkan db.Where("status = ?", status).Find(&histories).

Dengan adanya repository ini, service layer dapat dengan mudah berinteraksi dengan database tanpa perlu mengetahui detail implementasi query secara langsung. Repository ini mendukung pemrosesan histori transaksi dengan berbagai metode pencarian, yang nantinya akan digunakan dalam service dan controller untuk menangani permintaan pengguna terkait histori transaksi.

History_service.go

```
package service

import (
	"mikti-depublic/model/domain"
	"mikti-depublic/repository"
)

type HistoryService interface {
	CreateTransaction(transaction *domain.Transaction) error
	GetHistories() ([]domain.Transaction, error)
	GetHistoriesForUser(userID string) ([]domain.Transaction,
error)
	GetHistoryByID(id string) (*domain.Transaction, error)
	GetHistoryByStatus(status string) ([]domain.Transaction,
error)
}

func NewHistoryService(repository repository.HistoryRepository)
HistoryService {
	return NewHistoryServiceImpl(repository)
}
```

Kode di atas merupakan bagian dari service layer dalam arsitektur aplikasi Golang yang bertanggung jawab untuk menangani logika bisnis terkait histori transaksi. Interface HistoryService mendefinisikan beberapa metode utama yang memungkinkan interaksi dengan data transaksi, seperti pembuatan transaksi baru (CreateTransaction), pengambilan semua transaksi (GetHistories), pencarian transaksi berdasarkan pengguna (GetHistoriesForUser), pencarian transaksi berdasarkan ID (GetHistoryByID), serta pencarian berdasarkan status (GetHistoryByStatus).

Fungsi NewHistoryService bertindak sebagai konstruktor yang mengembalikan instance dari HistoryServiceImpl, dengan menerima repository.HistoryRepository sebagai dependensi. Service ini berfungsi sebagai jembatan antara controller dan repository, di mana data yang diambil dari repository diproses sebelum dikembalikan ke controller untuk ditampilkan kepada pengguna. Namun, kode ini belum menyertakan implementasi dari NewHistoryServiceImpl, sehingga service ini masih belum dapat digunakan secara langsung dan memerlukan tambahan implementasi lebih lanjut.

History_service_impl.go

```
package service

import (
	"mikti-depublic/model/domain"
	"mikti-depublic/repository"
)

type historyServiceImpl struct {
	repository repository.HistoryRepository
}

func NewHistoryServiceImpl(repository
repository.HistoryRepository) HistoryService {
	return &historyServiceImpl{repository: repository}
}
```

```
func (s *historyServiceImpl) CreateTransaction(transaction
*domain.Transaction) error {
	return s.repository.Create(transaction)
}

func (s *historyServiceImpl) GetHistories()
([]domain.Transaction, error) {
	return s.repository.FindAll()
}

func (s *historyServiceImpl) GetHistoriesForUser(id string)
([]domain.Transaction, error) {
	return s.repository.FindByUserID(id)
}

func (s *historyServiceImpl) GetHistoryByID(id string)
(*domain.Transaction, error) {
	return s.repository.FindByID(id)
}

func (s *historyServiceImpl) GetHistoryByStatus(status string)
([]domain.Transaction, error) {
	return s.repository.FindByStatus(status)
}
```

Kode di atas merupakan implementasi dari service layer dalam arsitektur aplikasi Golang yang bertanggung jawab untuk menangani logika bisnis terkait histori transaksi. Struct historyServiceImpl didefinisikan dengan atribut repository bertipe repository.HistoryRepository, yang memungkinkan service berinteraksi dengan repository untuk mengakses database. Fungsi NewHistoryServiceImpl bertindak sebagai konstruktor, yang mengembalikan instance dari historyServiceImpl dengan menyertakan repository sebagai dependensinya.

Beberapa metode utama dalam service ini antara lain CreateTransaction, yang berfungsi untuk menambahkan transaksi baru dengan memanggil metode Create pada

repository. Metode GetHistories digunakan untuk mengambil semua transaksi dalam database melalui FindAll, sedangkan GetHistoriesForUser memungkinkan pengambilan transaksi berdasarkan ID pengguna menggunakan FindByUserID. Selain itu, GetHistoryByID digunakan untuk mendapatkan transaksi berdasarkan ID transaksi, dan GetHistoryByStatus mengambil transaksi berdasarkan status tertentu.

Service ini berperan sebagai lapisan perantara antara controller dan repository, sehingga controller tidak perlu langsung berinteraksi dengan database. Dengan adanya service ini, pemisahan logika bisnis menjadi lebih terstruktur, meningkatkan fleksibilitas serta kemudahan dalam pengelolaan data transaksi dalam aplikasi.