```
go-project/
│
├── main.go
├── handler/
│   └── handler.go
├── controller/
│   └── controller.go
├── repository/
│   └── repository.go
├── model/
│   └── config.go
├── go.mod
├── go.sum
└── config/
    └── db_config.go
```

```
package repository

import (
	"context"
	"fmt"
	"go-project/model"
	"github.com/jackc/pgx/v4/pgxpool"
	"log"
)

// Repository interface defines methods for accessing service data
type Repository interface {
	GetServiceConfig() (*model.ServiceConfig, error)
	InsertServiceConfig(serviceConfig model.ServiceConfig) error
}

// repository struct implements the Repository interface
type repository struct {
	db *pgxpool.Pool
}

// NewRepository initializes and returns a new repository instance with a database
connection
func NewRepository(dbHost, dbPort, dbUser, dbPassword, dbName string) (Repository,
error) {
	// Create the DSN (Data Source Name) string for connecting to the PostgreSQL
database
	dsn := fmt.Sprintf("postgres://%s:%s@%s:%s/%s", dbUser, dbPassword, dbHost,
dbPort, dbName)
```

```
	// Create a connection pool to the PostgreSQL database
	db, err := pgxpool.Connect(context.Background(), dsn)
	if err != nil {
		log.Fatalf("Unable to connect to the database: %v", err)
		return nil, err
	}

	// Return a new repository instance with the database connection
	return &repository{db: db}, nil
}

// GetServiceConfig retrieves the service configuration from the database
func (r *repository) GetServiceConfig() (*model.ServiceConfig, error) {
	var serviceConfig model.ServiceConfig

	query := `SELECT service_name FROM service_config LIMIT 1`
	err := r.db.QueryRow(context.Background(),
query).Scan(&serviceConfig.ServiceName)
	if err != nil {
		return nil, err
	}

	return &serviceConfig, nil
}

// InsertServiceConfig inserts a new service configuration into the database
func (r *repository) InsertServiceConfig(serviceConfig model.ServiceConfig) error {
	query := `INSERT INTO service_config (service_name) VALUES ($1)`
	_, err := r.db.Exec(context.Background(), query, serviceConfig.ServiceName)
	if err != nil {
		return err
	}

	return nil
}


package controller

import (
	"context"
	"go-project/repository"
	"go-project/model"
)

// Controller interface defines methods for handling business logic
type Controller interface {
	GetServiceName(ctx context.Context) (string, error)
```

```
	CreateServiceConfig(ctx context.Context, name string) error
}

// controller struct implements the Controller interface
type controller struct {
	repo repository.Repository
}

// NewController initializes and returns a new controller instance
func NewController(repo repository.Repository) Controller {
	return &controller{
		repo: repo,
	}
}

// GetServiceName retrieves the service name from the repository
func (c *controller) GetServiceName(ctx context.Context) (string, error) {
	config, err := c.repo.GetServiceConfig()
	if err != nil {
		return "", err
	}
	return config.ServiceName, nil
}

// CreateServiceConfig creates a new service configuration
func (c *controller) CreateServiceConfig(ctx context.Context, name string) error {
	config := model.ServiceConfig{
		ServiceName: name,
	}

	return c.repo.InsertServiceConfig(config)
}



package handler

import (
	"context"
	"encoding/json"
	"net/http"
	"go-project/controller"
)

// Handler interface defines the methods the handler must implement
type Handler interface {
	GetServiceName(w http.ResponseWriter, r *http.Request)
	CreateServiceConfig(w http.ResponseWriter, r *http.Request)
```

```
}

// handler struct implements the Handler interface
type handler struct {
	controller controller.Controller
	ctx        context.Context
}

// NewHandler initializes and returns a new handler instance
func NewHandler(controller controller.Controller) (Handler, error) {
	return &handler{
		controller: controller,
		ctx:        context.Background(),
	}, nil
}

// GetServiceName sends the service name as a JSON response
func (h *handler) GetServiceName(w http.ResponseWriter, r *http.Request) {
	serviceName, err := h.controller.GetServiceName(h.ctx)
	if err != nil {
		http.Error(w, "Failed to get service name", http.StatusInternalServerError)
		return
	}

	w.Header().Set("Content-Type", "application/json")
	json.NewEncoder(w).Encode(map[string]string{
		"service_name": serviceName,
	})
}

// CreateServiceConfig handles the creation of a new service configuration
func (h *handler) CreateServiceConfig(w http.ResponseWriter, r *http.Request) {
	var req struct {
		ServiceName string `json:"service_name"`
	}

	if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
		http.Error(w, "Invalid request payload", http.StatusBadRequest)
		return
	}

	if err := h.controller.CreateServiceConfig(h.ctx, req.ServiceName); err != nil {
		http.Error(w, "Failed to create service config", http.StatusInternalServerError)
		return
	}

	w.WriteHeader(http.StatusCreated)
}
```

```
package main

import (
	"log"
	"net/http"
	"go-project/handler"
	"go-project/controller"
	"go-project/repository"
)

func main() {
	// Database connection parameters (these can be extracted from environment
variables in real applications)
	dbHost := "localhost"
	dbPort := "5432"
	dbUser := "postgres"
	dbPassword := "password"
	dbName := "go_project"

	// Initialize the repository with the database connection
	repo, err := repository.NewRepository(dbHost, dbPort, dbUser, dbPassword,
dbName)
	if err != nil {
		log.Fatalf("Error initializing repository: %v", err)
	}
	defer repo.(*repository.repository).db.Close() // Close the database connection when
the app stops

	// Initialize the controller with the repository
	ctrl := controller.NewController(repo)

	// Initialize the handler with the controller
	h, err := handler.NewHandler(ctrl)
	if err != nil {
		log.Fatalf("Error initializing handler: %v", err)
	}

	// Register the HTTP routes and handler functions
	http.HandleFunc("/service", h.GetServiceName)
	http.HandleFunc("/service/create", h.CreateServiceConfig)

	log.Println("Starting server on :8080")
	err = http.ListenAndServe(":8080", nil)
	if err != nil {
		log.Fatal(err)
	}
}
```