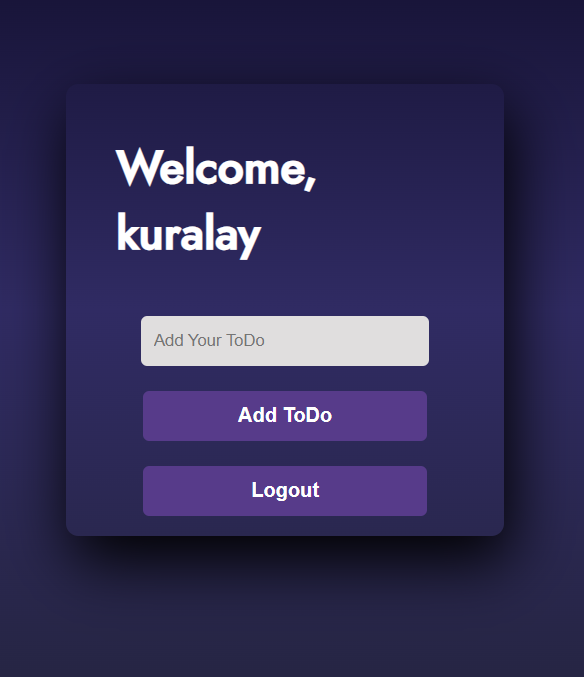
**Assignment 3**

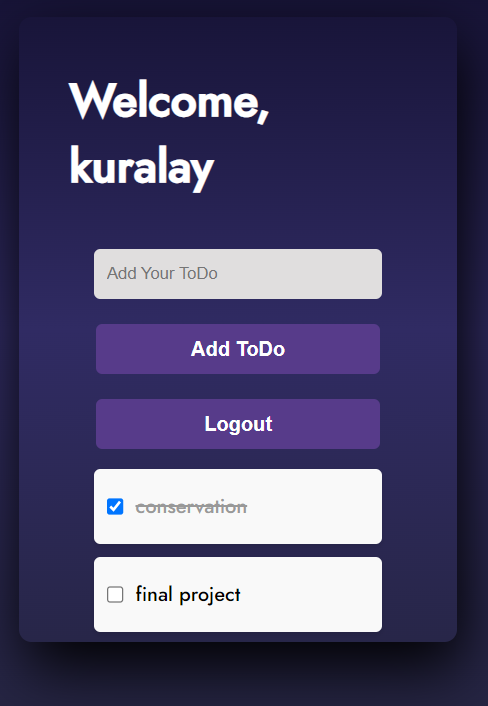
**REPORT**

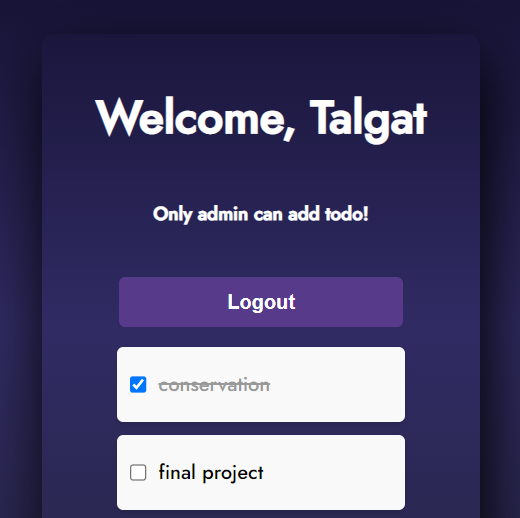
1. JWT Token Authorization

Exercise 1: Implement JWT Authentication Middleware

1. Create a simple web server using Golang.







1. Implement middleware that checks for a valid JWT token in the Authorization header for each incoming request.

package middlewares  
  
import (  
 "go-to-do/utils"  
  
 "github.com/gin-gonic/gin"  
)  
  
func IsAuthorized() gin.HandlerFunc {  
 return func(c \*gin.Context) {  
 cookie, err := c.Cookie("token")  
  
 if err != nil {  
 c.JSON(401, gin.H{"error": "unauthorized"})  
 c.Abort()  
 return  
 }  
  
 claims, err := utils.ParseToken(cookie)  
  
 if err != nil {  
 c.JSON(401, gin.H{"error": "unauthorized"})  
 c.Abort()  
 return  
 }  
  
 c.Set("role", claims.Role)  
 c.Next()  
 }  
}

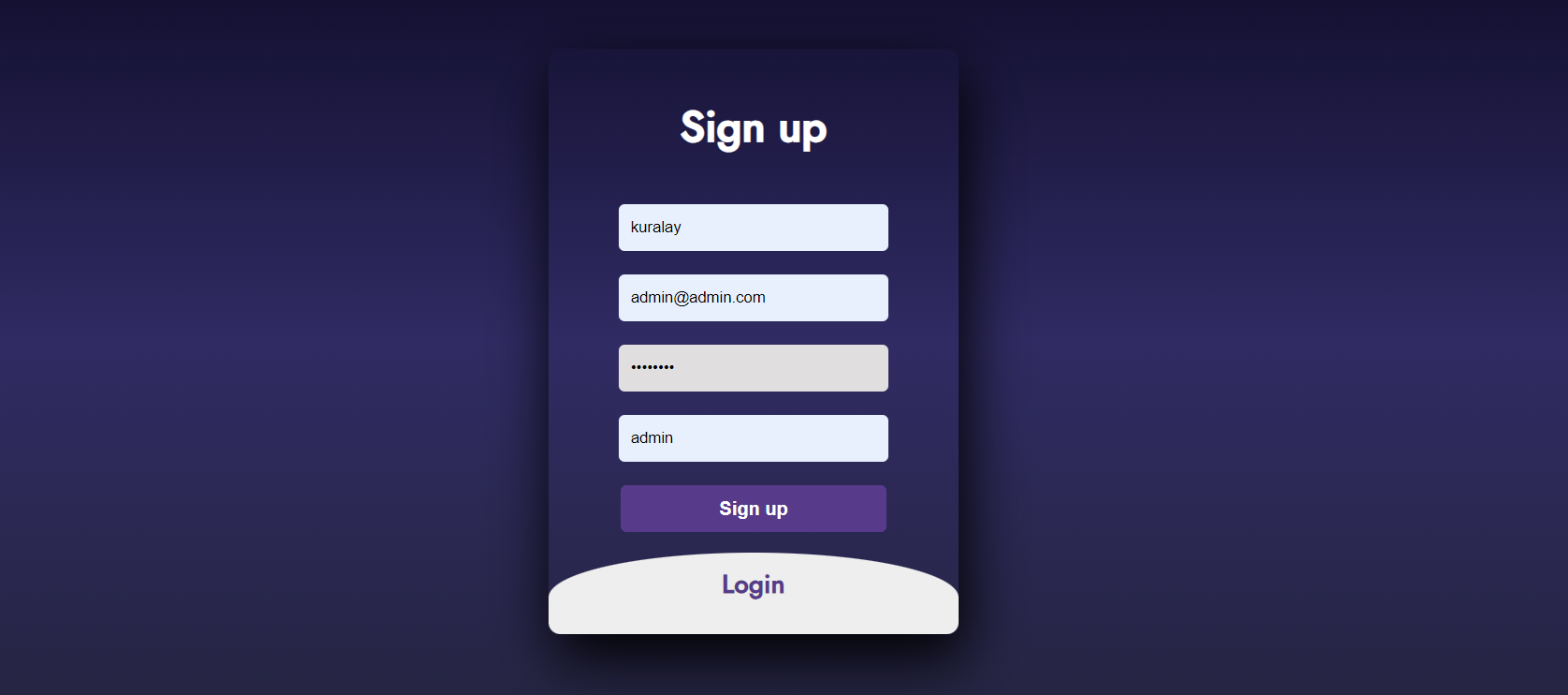
1. If the token is valid, allow the request to proceed; otherwise, return a 401 Unauthorized status.

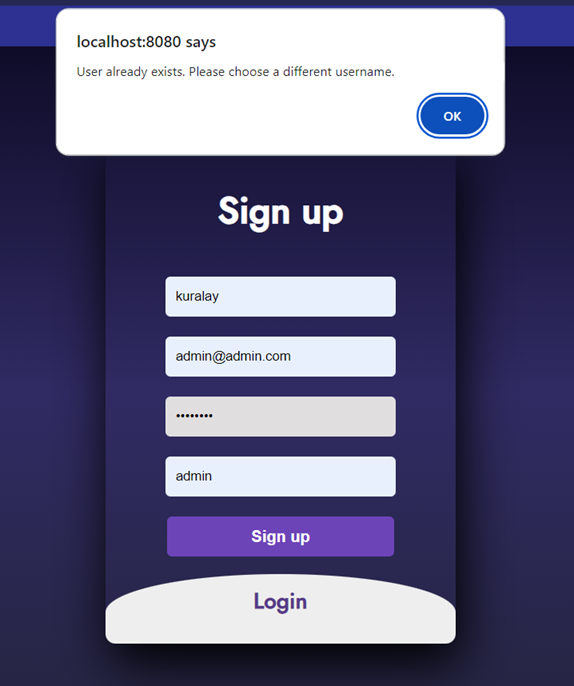
if err != nil {  
 c.JSON(401, gin.H{"error": "unauthorized"})  
 c.Abort()  
 return  
}

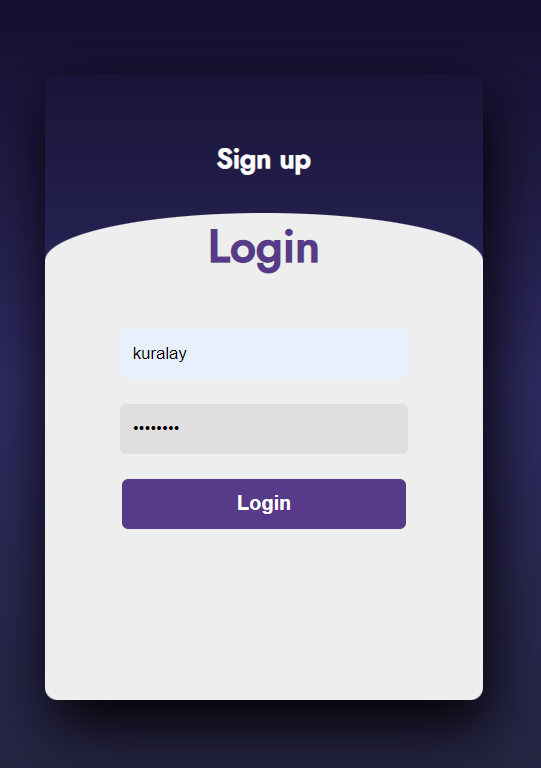
Exercise 2: User Sign-Up and Sign-In

1. Create endpoints for user registration (sign-up) and user login (sign-in).

func AuthRoutes(router \*gin.Engine) {  
 router.GET("/", controllers.WelcomePage)  
  
 router.POST("/add", controllers.AddToDo)  
  
 router.POST("/toggle", controllers.Toggle)  
  
 router.POST("/login", controllers.Login)  
  
 router.GET("/logout", controllers.Logout)  
  
 router.POST("/register", controllers.Register)  
}







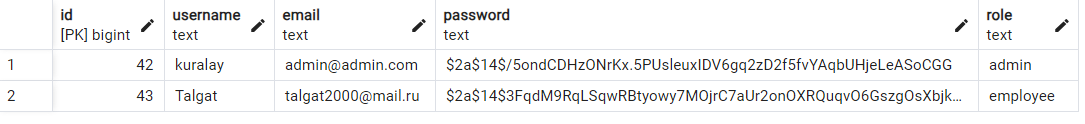
1. Use JWTs to authenticate users upon successful login and issue tokens.

func Register(c \*gin.Context) {  
 var user models.User  
  
 if err := c.ShouldBind(&user); err != nil {  
 c.JSON(400, gin.H{"error": err.Error()})  
 return  
 }  
  
 var existingUser models.User  
 if err := models.DB.Where("username = ?", user.Username).First(&existingUser).Error; err != nil {  
 if !errors.Is(err, gorm.ErrRecordNotFound) {  
 c.JSON(500, gin.H{"error": "database error"})  
 return  
 }  
 }  
  
 if existingUser.ID != 0 {  
 c.JSON(400, gin.H{"error": "user already exists"})  
 return  
 }  
  
 var errHash error  
 user.Password, errHash = utils.GenerateHashPassword(user.Password)  
  
 if errHash != nil {  
 c.JSON(500, gin.H{"error": "could not generate password hash"})  
 return  
 }  
  
 if err := models.DB.Create(&user).Error; err != nil {  
 c.JSON(500, gin.H{"error": "could not create user"})  
 return  
 }  
  
 loggedInUser = user  
  
 c.JSON(200, gin.H{"success": "user created"})  
 fmt.Printf("Sign up loggedInUser: %+v\n", loggedInUser)  
 fmt.Printf("Sign up user: %+v\n", user)  
  
 c.Redirect(http.*StatusSeeOther*, "/")  
}  
  
func Login(c \*gin.Context) {  
 var user models.User  
  
 if err := c.ShouldBindJSON(&user); err != nil {  
 c.JSON(400, gin.H{"error": err.Error()})  
 return  
 }  
  
 models.DB.Where("username = ?", user.Username).First(&loggedInUser)  
  
 if loggedInUser.ID == 0 {  
 c.JSON(400, gin.H{"error": "user does not exist"})  
 return  
 }  
  
 errHash := utils.CompareHashPassword(user.Password, loggedInUser.Password)  
  
 if !errHash {  
 c.JSON(400, gin.H{"error": "invalid password"})  
 return  
 }  
  
 fmt.Printf("Login loggedInUser: %+v\n", loggedInUser)  
 fmt.Printf("Login user: %+v\n", user)  
  
 expirationTime := time.Now().Add(5 \* time.*Minute*)  
  
 claims := &models.Claims{  
 Role: loggedInUser.Role,  
 StandardClaims: jwt.StandardClaims{  
 Subject: loggedInUser.Email,  
 ExpiresAt: expirationTime.Unix(),  
 },  
 }  
  
 token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)  
  
 tokenString, err := token.SignedString(jwtKey)  
  
 if err != nil {  
 c.JSON(500, gin.H{"error": "could not generate token"})  
 return  
 }  
  
 c.SetCookie("token", tokenString, int(expirationTime.Unix()), "/", "localhost", false, true)  
 c.JSON(200, gin.H{"success": "user logged in"})  
 c.Redirect(http.*StatusSeeOther*, "/")  
}

1. Store user credentials securely and hash passwords before storing.

func GenerateHashPassword(password string) (string, error) {  
 bytes, err := bcrypt.GenerateFromPassword([]byte(password), 14)  
 return string(bytes), err  
}

func CompareHashPassword(password, hash string) bool {  
 err := bcrypt.CompareHashAndPassword([]byte(hash), []byte(password))  
 return err == nil  
}



1. Concurrency and Parallelism

Exercise 1: Goroutines and Channels

1. Create a program that concurrently fetches data from multiple APIs.

In the code, the main() function creates a map called apiURLs that contains the URLs for different API endpoints. Then, it initializes a sync.WaitGroup and a results channel for managing concurrent requests and collecting results. For each URL in apiURLs, the program launches a goroutine that calls the fetchAPI() function. This function fetches data from the API and sends the results to the results channel. By using goroutines, the program can fetch data from multiple APIs concurrently.

var apiURLs = map[string]string{  
 "events": "https://shazam.p.rapidapi.com/shazam-events/list?artistId=73406786&l=en-US&from=2022-12-31&limit=50&offset=0",  
 "search": "https://shazam.p.rapidapi.com/search?term=kiss%20the%20rain&locale=en-US&offset=0&limit=5",  
 "autocomplete": "https://shazam.p.rapidapi.com/auto-complete?term=kiss%20the&locale=en-US",  
}

1. Use goroutines to initiate API requests.

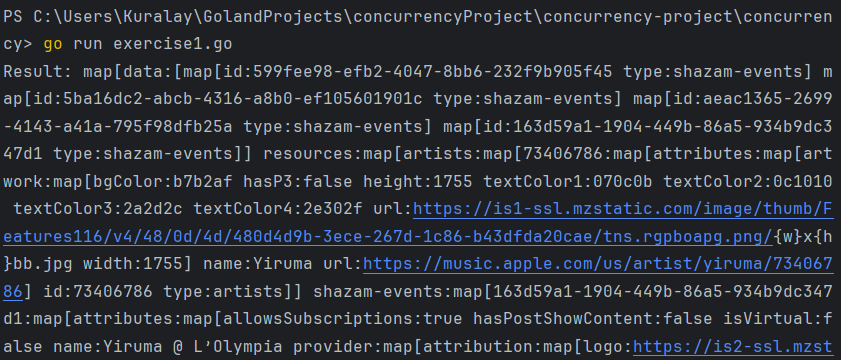
The fetchAPI() function is launched as a goroutine for each URL in the apiURLs map. The function takes a URL, headers, a results channel, and a sync.WaitGroup as parameters. It performs an API request and sends the response data to the results channel. The use of goroutines allows the program to handle multiple API requests concurrently, making the program more efficient.

for \_, url := range apiURLs {  
 wg.Add(1)  
 go fetchAPI(url, headers, results, &wg)  
}

1. Use channels to collect and merge the results from different API calls.

In the main() function, a channel called results is initialized to collect the results from the API requests. Each goroutine sends the data it fetches to this channel. The main() function then reads from the results channel in a loop, printing each API response to the console. By using a channel, the program can effectively collect and merge the results from multiple API requests.

for responseData := range results {  
 fmt.Printf("Result: %+v\n", responseData)  
}

…

The provided result contains information about events and venues associated with the artist Yiruma

Exercise 2: Parallel Processing

1. Write a program that reads a list of URLs from a file and downloads the content from each URL concurrently.

The program reads a list of URLs from a file named urls.txt using the function readURLsFromFile. This function returns a list of URLs:

filename := "urls.txt"  
urls, err := readURLsFromFile(filename)  
if err != nil {  
 log.Fatalf("Error reading URLs from file %s: %s", filename, err)  
}

The program uses goroutines to download content from each URL concurrently. Each URL is passed to the downloadFile function in a separate goroutine:

for \_, url := range urls {  
 workerPool <- ""  
 go downloadFile(url, workerPool)  
}

1. Utilize a worker pool to limit the number of simultaneous downloads.

The program creates a worker pool using a channel (workerPool) with a maximum capacity (maxWorkers) to limit the number of simultaneous downloads:

const *maxWorkers* = 5

workerPool := make(chan string, *maxWorkers*)

Before starting a download, the program waits for a free spot in the worker pool:

for \_, url := range urls {  
 workerPool <- ""  
 go downloadFile(url, workerPool)  
}

1. Ensure proper synchronization to avoid race conditions.

The program ensures proper synchronization using a defer block in the downloadFile function to release a spot in the worker pool after each download completes:

func downloadFile(url string, workerPool chan string) {  
 defer func() {  
 <-workerPool  
 }()

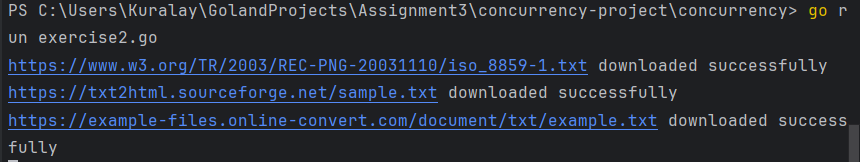
//…

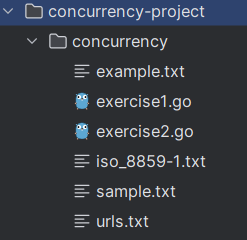
}

Additionally, the program waits for all goroutines to complete by reading from the worker pool:

for i := 0; i < cap(workerPool); i++ {  
 <-workerPool  
}

When you run the code, it will download the content of the three text files from the provided URLs concurrently:



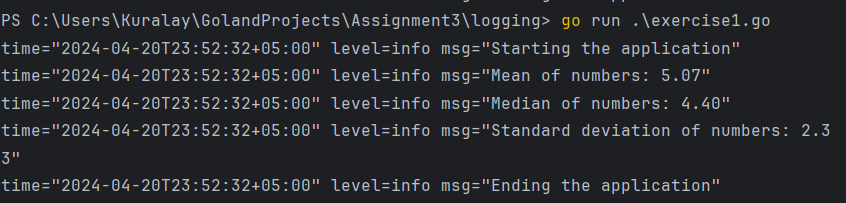


1. Logging

Exercise 1: Basic Logging

1. Develop a simple application that performs some operations.

The program performs statistical operations such as calculating the mean, median, and standard deviation of an array of numbers.



numbers := []float64{3.4, 2.1, 5.8, 3.2, 7.6, 9.0, 4.4}  
mean := calculateMean(numbers, logger)  
median := calculateMedian(numbers, logger)  
stdDev := calculateStdDev(numbers, mean, logger)

1. Implement logging to track the flow of execution and any errors that occur.

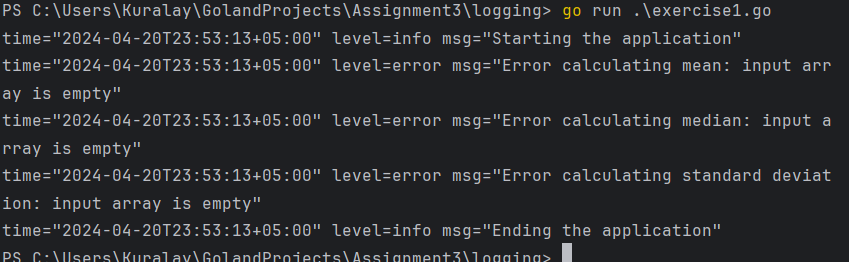
The program uses logrus for logging, which provides structured logging with different levels (Info, Error).

logger.Info("Starting the application")  
logger.Infof("Mean of numbers: %.2f", mean)  
logger.Infof("Median of numbers: %.2f", median)  
logger.Infof("Standard deviation of numbers: %.2f", stdDev)

logger.Info("Ending the application")

Logging Errors: If there is an error during execution, such as trying to calculate statistics on an empty array, the program logs an error using logrus.Errorf:

if len(numbers) == 0 {  
 return 0, fmt.Errorf("input array is empty")  
}

Error Handling: The program checks for errors during execution and logs them using logrus:

logger.Errorf("Error calculating mean: %v", err)

logger.Errorf("Error calculating median: %v", err)

logger.Errorf("Error calculating standard deviation: %v", err)

In this example, if an error occurs while calculating the mean, the program logs the error using logrus.Errorf with a descriptive message.

mean, err := calculateMean(numbers, logger)  
if err != nil {  
 logger.Errorf("Error calculating mean: %v", err)  
}

1. Utilize the standard Go logging package or popular logging libraries like logrus.

The program uses the popular logrus library for logging, providing flexibility in formatting and levels of logging.

logger.SetFormatter(&logrus.TextFormatter{})

Exercise 2: Structured Logging

1. Extend the previous application to log events with structured data (e.g., JSON format).

The code uses the logrus library for logging events in a structured JSON format:

logger := logrus.New()  
logger.SetFormatter(&logrus.JSONFormatter{})

1. Include relevant metadata such as timestamps, request IDs, or user IDs in the log entries.

Logs include relevant metadata such as timestamps (timestamp), request IDs (request\_id), and user IDs (user\_id) in the log entries. This helps provide context and trace requests:

logger.WithFields(logrus.Fields{  
 "timestamp": time.Now().Format(time.*RFC3339*),  
 "request\_id": requestID,  
 "user\_id": userID,  
}).Info("Starting the application")

These metadata fields are included in the logs for each operation, which aids in analysis and debugging.

1. Demonstrate how structured logging aids in analysis and troubleshooting

The program demonstrates how structured logging aids in analysis and troubleshooting by including information about the results of operations (e.g., mean, median, and standard deviation) and errors.

Error logging is handled with detailed information about problems, such as empty arrays:

if len(numbers) == 0 {  
 err := fmt.Errorf("input array is empty")  
 logError(err, logger, requestID, userID, "Error calculating mean")  
 return 0, err  
}

Using structured logging allows for grouping logs by requests and users, which aids in analysis and troubleshooting

