# Prompts and Responses

## Prompt:

I want to write an web application which will be using Golang as backend language and plain html css JS and AJAX for frontend . I want you to set this chat for providing suggestions for the project. I want to use MYSQL as database, and gin framework for api development and you can use the native sql queries for db operation. Problem statement is as below: database user root, database name hackathon, db password is root.   
1. Api to login and signup, with user details, name, designation, experience, skills (comma seprated)   
2. Api to create a create a proposal and assign it to a coordinator with deadline.   
3. Api to create subtasks and publish subtask with topic name, description, skiils required, subtasks should have submission date.   
4. Api upload subtasks and track status.   
5. Api to review the subtasks within the timeline.   
6. Api to download the presentation slides.   
  
Give me the database tables to be created with proper relation ships and queries so that I can create it in my database and then give me the go code. Also make sure there are no CORS error when I consume these APIs in frontend.

## Response:

### Database Design

You need multiple tables with relationships for users, proposals, subtasks, and submissions. Here's a design for your database tables and SQL queries to create them:

#### 1. ****Users Table****

Stores information about users including their designation and skills.

sql

Copy code

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

designation VARCHAR(255),

experience INT,

skills TEXT, -- Comma-separated skills

email VARCHAR(255) UNIQUE NOT NULL,

password VARCHAR(255) NOT NULL -- Use hashed passwords

);

#### 2. ****Proposals Table****

Tracks proposals and assigns them to coordinators.

sql

Copy code

CREATE TABLE proposals (

id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

description TEXT,

coordinator\_id INT, -- Foreign key to users table

deadline DATE NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (coordinator\_id) REFERENCES users(id)

);

#### 3. ****Subtasks Table****

Each proposal can have multiple subtasks with a topic name, required skills, and submission date.

sql

Copy code

CREATE TABLE subtasks (

id INT AUTO\_INCREMENT PRIMARY KEY,

proposal\_id INT, -- Foreign key to proposals table

topic\_name VARCHAR(255) NOT NULL,

description TEXT,

skills\_required TEXT, -- Comma-separated skills

submission\_date DATE,

status ENUM('pending', 'completed') DEFAULT 'pending',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (proposal\_id) REFERENCES proposals(id)

);

#### 4. ****Subtask Submissions Table****

Tracks the uploaded submissions and their status.

sql

Copy code

CREATE TABLE submissions (

id INT AUTO\_INCREMENT PRIMARY KEY,

subtask\_id INT, -- Foreign key to subtasks table

user\_id INT, -- Foreign key to users table (who uploaded the task)

file\_path VARCHAR(255) NOT NULL,

submission\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

review\_status ENUM('pending', 'approved', 'rejected') DEFAULT 'pending',

reviewer\_id INT, -- Foreign key to users table

review\_date TIMESTAMP,

FOREIGN KEY (subtask\_id) REFERENCES subtasks(id),

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (reviewer\_id) REFERENCES users(id)

);

### Basic SQL Queries

1. **Create a User (Signup)**

sql

Copy code

INSERT INTO users (name, designation, experience, skills, email, password)

VALUES ('John Doe', 'Developer', 5, 'Go,Python', 'john@example.com', 'hashed\_password');

1. **Login Query**

sql

Copy code

SELECT \* FROM users WHERE email = 'john@example.com' AND password = 'hashed\_password';

1. **Create a Proposal**

sql

Copy code

INSERT INTO proposals (title, description, coordinator\_id, deadline)

VALUES ('Proposal Title', 'Proposal Description', 1, '2024-10-01');

1. **Create a Subtask**

sql

Copy code

INSERT INTO subtasks (proposal\_id, topic\_name, description, skills\_required, submission\_date)

VALUES (1, 'Subtask Topic', 'Subtask Description', 'Go,HTML', '2024-09-30');

1. **Upload a Subtask Submission**

sql

Copy code

INSERT INTO submissions (subtask\_id, user\_id, file\_path)

VALUES (1, 2, '/path/to/file');

1. **Review a Subtask**

sql

Copy code

UPDATE submissions

SET review\_status = 'approved', reviewer\_id = 1, review\_date = NOW()

WHERE id = 1;

### Golang Code Using Gin Framework

Below is the basic structure for the Gin API routes. This code doesn't handle every edge case but should provide a good starting point:

1. **Setup Router & CORS Middleware**

go

Copy code

package main

import (

"github.com/gin-gonic/gin"

"github.com/gin-contrib/cors"

)

func main() {

r := gin.Default()

// CORS middleware to avoid CORS errors

r.Use(cors.Default())

r.POST("/signup", SignupHandler)

r.POST("/login", LoginHandler)

r.POST("/proposal", CreateProposalHandler)

r.POST("/subtask", CreateSubtaskHandler)

r.POST("/subtask/upload", UploadSubtaskHandler)

r.POST("/subtask/review", ReviewSubtaskHandler)

r.Run(":8080") // Start the server on port 8080

}

1. **Signup & Login Handlers**

go

Copy code

package main

import (

"github.com/gin-gonic/gin"

"net/http"

"database/sql"

\_ "github.com/go-sql-driver/mysql"

)

var db \*sql.DB

func init() {

var err error

db, err = sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/hackathon")

if err != nil {

panic(err)

}

}

func SignupHandler(c \*gin.Context) {

var user User

if err := c.ShouldBindJSON(&user); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

query := "INSERT INTO users (name, designation, experience, skills, email, password) VALUES (?, ?, ?, ?, ?, ?)"

\_, err := db.Exec(query, user.Name, user.Designation, user.Experience, user.Skills, user.Email, user.Password)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create user"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Signup successful"})

}

func LoginHandler(c \*gin.Context) {

var creds LoginCredentials

if err := c.ShouldBindJSON(&creds); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

query := "SELECT id FROM users WHERE email = ? AND password = ?"

var userID int

err := db.QueryRow(query, creds.Email, creds.Password).Scan(&userID)

if err != nil {

c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid credentials"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Login successful", "user\_id": userID})

}

The SignupHandler inserts a new user into the database, while the LoginHandler checks user credentials.

1. **Create Proposal Handler**

go

Copy code

func CreateProposalHandler(c \*gin.Context) {

var proposal Proposal

if err := c.ShouldBindJSON(&proposal); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

query := "INSERT INTO proposals (title, description, coordinator\_id, deadline) VALUES (?, ?, ?, ?)"

\_, err := db.Exec(query, proposal.Title, proposal.Description, proposal.CoordinatorID, proposal.Deadline)

if err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create proposal"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Proposal created successfully"})

}

### Handling CORS in AJAX

Make sure your AJAX requests include the appropriate headers:

javascript

Copy code

$.ajax({

url: "http://localhost:8080/login",

type: "POST",

contentType: "application/json",

data: JSON.stringify({

email: "john@example.com",

password: "hashed\_password"

}),

success: function(response) {

console.log(response);

},

error: function(error) {

console.error(error);

}

});

Let me know if you need help with specific parts of the code!

## 

## Prompt:

Can you write me all handlers with all structs/models used along with database queries in the functions. You can write handlers in one file, models in one file and database functions in another file.

## Response:

Here’s a modular approach for writing your Go application using the Gin framework. The code will be divided into three files:

1. handlers.go (contains all the HTTP handlers)
2. models.go (defines all the models and structs)
3. db.go (handles database operations)

Let’s walk through the code for each file.

### 1. ****handlers.go****

This file will contain all your HTTP handlers for the routes defined in the Gin framework.

go

Copy code

package main

import (

"net/http"

"time"

"github.com/gin-gonic/gin"

)

// SignupHandler handles user registration

func SignupHandler(c \*gin.Context) {

var user User

if err := c.ShouldBindJSON(&user); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

if err := CreateUser(user); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create user"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Signup successful"})

}

// LoginHandler handles user login

func LoginHandler(c \*gin.Context) {

var creds LoginCredentials

if err := c.ShouldBindJSON(&creds); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

userID, err := AuthenticateUser(creds)

if err != nil {

c.JSON(http.StatusUnauthorized, gin.H{"error": "Invalid credentials"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Login successful", "user\_id": userID})

}

// CreateProposalHandler handles the creation of proposals

func CreateProposalHandler(c \*gin.Context) {

var proposal Proposal

if err := c.ShouldBindJSON(&proposal); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

if err := CreateProposal(proposal); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create proposal"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Proposal created successfully"})

}

// CreateSubtaskHandler handles subtask creation

func CreateSubtaskHandler(c \*gin.Context) {

var subtask Subtask

if err := c.ShouldBindJSON(&subtask); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

if err := CreateSubtask(subtask); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create subtask"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Subtask created successfully"})

}

// UploadSubtaskHandler handles the uploading of subtasks

func UploadSubtaskHandler(c \*gin.Context) {

var submission Submission

if err := c.ShouldBindJSON(&submission); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

submission.SubmissionDate = time.Now()

if err := UploadSubtask(submission); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to upload subtask"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Subtask uploaded successfully"})

}

// ReviewSubtaskHandler handles the review of subtasks

func ReviewSubtaskHandler(c \*gin.Context) {

var review SubmissionReview

if err := c.ShouldBindJSON(&review); err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})

return

}

review.ReviewDate = time.Now()

if err := ReviewSubtask(review); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to review subtask"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Subtask reviewed successfully"})

}

### 2. ****models.go****

This file contains the models and structs that represent the entities in your application, like User, Proposal, Subtask, etc.

go

Copy code

package main

import "time"

// User struct represents a user in the system

type User struct {

ID int `json:"id"`

Name string `json:"name"`

Designation string `json:"designation"`

Experience int `json:"experience"`

Skills string `json:"skills"` // Comma-separated list

Email string `json:"email"`

Password string `json:"password"`

}

// LoginCredentials struct for user login

type LoginCredentials struct {

Email string `json:"email"`

Password string `json:"password"`

}

// Proposal struct represents a proposal

type Proposal struct {

ID int `json:"id"`

Title string `json:"title"`

Description string `json:"description"`

CoordinatorID int `json:"coordinator\_id"`

Deadline string `json:"deadline"` // YYYY-MM-DD format

}

// Subtask struct represents a subtask

type Subtask struct {

ID int `json:"id"`

ProposalID int `json:"proposal\_id"`

TopicName string `json:"topic\_name"`

Description string `json:"description"`

SkillsRequired string `json:"skills\_required"` // Comma-separated list

SubmissionDate string `json:"submission\_date"` // YYYY-MM-DD format

Status string `json:"status"`

}

// Submission struct represents a subtask submission

type Submission struct {

ID int `json:"id"`

SubtaskID int `json:"subtask\_id"`

UserID int `json:"user\_id"`

FilePath string `json:"file\_path"`

SubmissionDate time.Time `json:"submission\_date"`

}

// SubmissionReview struct for reviewing a submission

type SubmissionReview struct {

SubmissionID int `json:"submission\_id"`

ReviewerID int `json:"reviewer\_id"`

ReviewStatus string `json:"review\_status"`

ReviewDate time.Time `json:"review\_date"`

}

### 3. ****db.go****

This file handles all database interactions, such as inserting, querying, and updating records.

go

Copy code

package main

import (

"database/sql"

\_ "github.com/go-sql-driver/mysql"

"errors"

)

var db \*sql.DB

// Initialize database connection

func init() {

var err error

db, err = sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/hackathon")

if err != nil {

panic(err)

}

}

// CreateUser inserts a new user into the database

func CreateUser(user User) error {

query := "INSERT INTO users (name, designation, experience, skills, email, password) VALUES (?, ?, ?, ?, ?, ?)"

\_, err := db.Exec(query, user.Name, user.Designation, user.Experience, user.Skills, user.Email, user.Password)

return err

}

// AuthenticateUser checks if the user's credentials are correct

func AuthenticateUser(creds LoginCredentials) (int, error) {

query := "SELECT id FROM users WHERE email = ? AND password = ?"

var userID int

err := db.QueryRow(query, creds.Email, creds.Password).Scan(&userID)

if err != nil {

return 0, errors.New("invalid credentials")

}

return userID, nil

}

// CreateProposal inserts a new proposal into the database

func CreateProposal(proposal Proposal) error {

query := "INSERT INTO proposals (title, description, coordinator\_id, deadline) VALUES (?, ?, ?, ?)"

\_, err := db.Exec(query, proposal.Title, proposal.Description, proposal.CoordinatorID, proposal.Deadline)

return err

}

// CreateSubtask inserts a new subtask into the database

func CreateSubtask(subtask Subtask) error {

query := "INSERT INTO subtasks (proposal\_id, topic\_name, description, skills\_required, submission\_date, status) VALUES (?, ?, ?, ?, ?, ?)"

\_, err := db.Exec(query, subtask.ProposalID, subtask.TopicName, subtask.Description, subtask.SkillsRequired, subtask.SubmissionDate, subtask.Status)

return err

}

// UploadSubtask inserts a subtask submission into the database

func UploadSubtask(submission Submission) error {

query := "INSERT INTO submissions (subtask\_id, user\_id, file\_path, submission\_date) VALUES (?, ?, ?, ?)"

\_, err := db.Exec(query, submission.SubtaskID, submission.UserID, submission.FilePath, submission.SubmissionDate)

return err

}

// ReviewSubtask updates the status of a subtask submission

func ReviewSubtask(review SubmissionReview) error {

query := "UPDATE submissions SET review\_status = ?, reviewer\_id = ?, review\_date = ? WHERE id = ?"

\_, err := db.Exec(query, review.ReviewStatus, review.ReviewerID, review.ReviewDate, review.SubmissionID)

return err

}

### Putting it all together

1. **handlers.go** defines the handlers for each API endpoint.
2. **models.go** defines the data models (structs) for users, proposals, subtasks, and submissions.
3. **db.go** contains functions that perform database operations like creating users, proposals, and subtasks, and uploading and reviewing submissions.

### Server Setup (in main.go)

go

Copy code

package main

import (

"github.com/gin-gonic/gin"

"github.com/gin-contrib/cors"

)

func main() {

r := gin.Default()

// CORS middleware to avoid CORS errors

r.Use(cors.Default())

// API routes

r.POST("/signup", SignupHandler)

r.POST("/login", LoginHandler)

r.POST("/proposal", CreateProposalHandler)

r.POST("/subtask", CreateSubtaskHandler)

r.POST("/subtask/upload", UploadSubtaskHandler)

r.POST("/subtask/review", ReviewSubtaskHandler)

r.Run(":8080") // Start the server on port 8080

}

This modular approach will help you maintain and scale your project efficiently. Let me know if you need further clarification or modifications!

## Prompt: can you write migration files as well and update my main function to use those migrations. Create table queries and all you already know.

## Response:

Sure! To integrate the database migration into your Go project, you can use raw SQL execution within the migration functions.

### Migrations Directory

Let's create a new directory named migrations and have a single migration file in it for now. We’ll later integrate this into the main.go file.

#### ****migrations/2024\_09\_27\_create\_tables.go****

This migration file will contain the necessary SQL statements to create the required tables.

go

Copy code

package migrations

import (

"database/sql"

"fmt"

)

// RunMigrations will execute all database migrations

func RunMigrations(db \*sql.DB) error {

if err := createUsersTable(db); err != nil {

return err

}

if err := createProposalsTable(db); err != nil {

return err

}

if err := createSubtasksTable(db); err != nil {

return err

}

if err := createSubmissionsTable(db); err != nil {

return err

}

fmt.Println("Migrations completed successfully!")

return nil

}

func createUsersTable(db \*sql.DB) error {

query := `

CREATE TABLE IF NOT EXISTS users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

designation VARCHAR(255),

experience INT,

skills TEXT,

email VARCHAR(255) UNIQUE NOT NULL,

password VARCHAR(255) NOT NULL

);

`

\_, err := db.Exec(query)

if err != nil {

return fmt.Errorf("error creating users table: %v", err)

}

fmt.Println("Users table created.")

return nil

}

func createProposalsTable(db \*sql.DB) error {

query := `

CREATE TABLE IF NOT EXISTS proposals (

id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255) NOT NULL,

description TEXT,

coordinator\_id INT,

deadline DATE NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (coordinator\_id) REFERENCES users(id)

);

`

\_, err := db.Exec(query)

if err != nil {

return fmt.Errorf("error creating proposals table: %v", err)

}

fmt.Println("Proposals table created.")

return nil

}

func createSubtasksTable(db \*sql.DB) error {

query := `

CREATE TABLE IF NOT EXISTS subtasks (

id INT AUTO\_INCREMENT PRIMARY KEY,

proposal\_id INT,

topic\_name VARCHAR(255) NOT NULL,

description TEXT,

skills\_required TEXT,

submission\_date DATE,

status ENUM('pending', 'completed') DEFAULT 'pending',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (proposal\_id) REFERENCES proposals(id)

);

`

\_, err := db.Exec(query)

if err != nil {

return fmt.Errorf("error creating subtasks table: %v", err)

}

fmt.Println("Subtasks table created.")

return nil

}

func createSubmissionsTable(db \*sql.DB) error {

query := `

CREATE TABLE IF NOT EXISTS submissions (

id INT AUTO\_INCREMENT PRIMARY KEY,

subtask\_id INT,

user\_id INT,

file\_path VARCHAR(255) NOT NULL,

submission\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

review\_status ENUM('pending', 'approved', 'rejected') DEFAULT 'pending',

reviewer\_id INT,

review\_date TIMESTAMP,

FOREIGN KEY (subtask\_id) REFERENCES subtasks(id),

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (reviewer\_id) REFERENCES users(id)

);

`

\_, err := db.Exec(query)

if err != nil {

return fmt.Errorf("error creating submissions table: %v", err)

}

fmt.Println("Submissions table created.")

return nil

}

### Update the Main Function

Now, we will update the main.go file to execute the migrations before the server starts. We will also ensure that the database connection is established before running the migrations.

#### ****main.go****

go

Copy code

package main

import (

"log"

"github.com/gin-gonic/gin"

"github.com/gin-contrib/cors"

"database/sql"

"migrations"

\_ "github.com/go-sql-driver/mysql"

)

func main() {

// Initialize the database

db, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/hackathon")

if err != nil {

log.Fatal("Failed to connect to the database: ", err)

}

// Run migrations

err = migrations.RunMigrations(db)

if err != nil {

log.Fatal("Migration failed: ", err)

}

defer db.Close()

r := gin.Default()

// CORS middleware to avoid CORS errors

r.Use(cors.Default())

// API routes

r.POST("/signup", SignupHandler)

r.POST("/login", LoginHandler)

r.POST("/proposal", CreateProposalHandler)

r.POST("/subtask", CreateSubtaskHandler)

r.POST("/subtask/upload", UploadSubtaskHandler)

r.POST("/subtask/review", ReviewSubtaskHandler)

// Start the server

r.Run(":8080")

}

### Directory Structure

Your project directory should now look something like this:

lua

Copy code

/your-project-directory

|-- /migrations

|-- 2024\_09\_27\_create\_tables.go

|-- db.go

|-- handlers.go

|-- models.go

|-- main.go

### Explanation

1. **Migration Execution**:
   * The function RunMigrations inside migrations/2024\_09\_27\_create\_tables.go will be called from main.go.
   * It sequentially calls the individual functions for creating the tables: createUsersTable, createProposalsTable, createSubtasksTable, and createSubmissionsTable.
   * The migrations ensure that if a table already exists, it will not be recreated, avoiding errors.
2. **Error Handling**:
   * If there's any error in establishing the database connection or running the migrations, the application will log the error and exit.
3. **Main Function**:
   * Before starting the HTTP server, we connect to the database and run the migrations. Once the migrations are successful, the HTTP server (Gin framework) will be started on port 8080.
4. **Modularity**:
   * The migration logic is separated from the application logic, making it easier to maintain and extend. You can add new migrations by creating new files in the migrations directory, ensuring that future database changes are applied in sequence.

This setup ensures that your database schema is always up to date when the application starts. Let me know if you need further assistance or modifications!

## Prompt:

Give me the curl command for all APIS you create above.

## Response:

Here are the curl commands for each of the APIs in your project:

### 1. ****Signup API****

#### Endpoint: POST /signup

bash

Copy code

curl -X POST http://localhost:8080/signup \

-H "Content-Type: application/json" \

-d '{

"name": "John Doe",

"designation": "Developer",

"experience": 5,

"skills": "Golang, MySQL",

"email": "johndoe@example.com",

"password": "password123"

}'

### 2. ****Login API****

#### Endpoint: POST /login

bash

Copy code

curl -X POST http://localhost:8080/login \

-H "Content-Type: application/json" \

-d '{

"email": "johndoe@example.com",

"password": "password123"

}'

### 3. ****Create Proposal API****

#### Endpoint: POST /proposal

bash

Copy code

curl -X POST http://localhost:8080/proposal \

-H "Content-Type: application/json" \

-d '{

"title": "New Web Application",

"description": "A web app project for hackathon",

"coordinator\_id": 1,

"deadline": "2024-12-31"

}'

### 4. ****Create Subtask API****

#### Endpoint: POST /subtask

bash

Copy code

curl -X POST http://localhost:8080/subtask \

-H "Content-Type: application/json" \

-d '{

"proposal\_id": 1,

"topic\_name": "Frontend Development",

"description": "Develop the frontend using HTML, CSS, and JavaScript",

"skills\_required": "HTML, CSS, JavaScript",

"submission\_date": "2024-11-15"

}'

### 5. ****Upload Subtask API****

#### Endpoint: POST /subtask/upload

This assumes you're uploading a file along with the request.

bash

Copy code

curl -X POST http://localhost:8080/subtask/upload \

-F "subtask\_id=1" \

-F "user\_id=1" \

-F "file=@/path/to/subtask/file.zip"

Replace /path/to/subtask/file.zip with the actual path of the file you are uploading.

### 6. ****Review Subtask API****

#### Endpoint: POST /subtask/review

bash

Copy code

curl -X POST http://localhost:8080/subtask/review \

-H "Content-Type: application/json" \

-d '{

"subtask\_id": 1,

"reviewer\_id": 2,

"review\_status": "approved"

}'

### 7. ****Download Presentation Slides API (If needed)****

#### Endpoint: GET /subtask/download

To download the file, if you plan to implement a file download endpoint:

bash

Copy code

curl -X GET http://localhost:8080/subtask/download?subtask\_id=1 \

-o presentation.zip

This will save the downloaded file as presentation.zip.

### Notes:

* **Replace IDs**: Make sure to replace the IDs (e.g., user\_id, subtask\_id, coordinator\_id) with the correct values as per your database.
* **File Upload**: For file uploads, make sure the path to the file is correct and the server is ready to handle file uploads.
* **Modify the Requests as Needed**: Depending on your request body structure, adjust the parameters.

Let me know if you need any more details or adjustments!

## Prompt:

Now can you add one more handler that can show me all proposals where I was coordinator and what is the status of it, what are the subtasksk I approved for that It is kind of reporting and tracking API.

## Response:

To add a reporting and tracking API that shows all proposals where a specific user is the coordinator, along with the subtasks they have approved, we can do the following:

### Steps:

1. **Modify handlers.go** to add the new handler.
2. **Write the database query** to fetch all proposals for which the user is the coordinator, along with the approved subtasks.

### API Endpoint:

**URL**: GET /coordinator/proposals/:userId

This endpoint will accept the userId (coordinator's ID) as a URL parameter and return a list of proposals where the user is the coordinator. For each proposal, it will include a list of approved subtasks.

### SQL Query Explanation:

We will join the proposals table and subtasks table, filter by coordinator\_id in proposals, and check the status and review\_status in subtasks and submissions.

### 1. Modify handlers.go

Here is the handler code that will handle the /coordinator/proposals/:userId request.

go

Copy code

package handlers

import (

"net/http"

"database/sql"

"github.com/gin-gonic/gin"

"log"

)

type SubtaskReport struct {

SubtaskID int `json:"subtask\_id"`

TopicName string `json:"topic\_name"`

Description string `json:"description"`

SkillsRequired string `json:"skills\_required"`

SubmissionDate string `json:"submission\_date"`

Status string `json:"status"`

ReviewerID int `json:"reviewer\_id"`

ReviewStatus string `json:"review\_status"`

}

type ProposalReport struct {

ProposalID int `json:"proposal\_id"`

Title string `json:"title"`

Description string `json:"description"`

Deadline string `json:"deadline"`

ApprovedSubtasks []SubtaskReport `json:"approved\_subtasks"`

}

// GetProposalsWithApprovedSubtasks handles the tracking and reporting request

func GetProposalsWithApprovedSubtasks(c \*gin.Context, db \*sql.DB) {

userID := c.Param("userId")

// SQL query to fetch all proposals where the user is the coordinator and fetch all approved subtasks

query := `

SELECT

p.id as proposal\_id, p.title, p.description, p.deadline,

s.id as subtask\_id, s.topic\_name, s.description, s.skills\_required, s.submission\_date, s.status,

sub.reviewer\_id, sub.review\_status

FROM proposals p

LEFT JOIN subtasks s ON p.id = s.proposal\_id

LEFT JOIN submissions sub ON s.id = sub.subtask\_id

WHERE p.coordinator\_id = ? AND sub.review\_status = 'approved';

`

rows, err := db.Query(query, userID)

if err != nil {

log.Println("Error executing query: ", err)

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to retrieve proposals"})

return

}

defer rows.Close()

proposalsMap := make(map[int]\*ProposalReport)

for rows.Next() {

var proposalID, subtaskID, reviewerID int

var title, proposalDescription, deadline, topicName, subtaskDescription, skillsRequired, submissionDate, status, reviewStatus string

if err := rows.Scan(&proposalID, &title, &proposalDescription, &deadline, &subtaskID, &topicName, &subtaskDescription, &skillsRequired, &submissionDate, &status, &reviewerID, &reviewStatus); err != nil {

log.Println("Error scanning rows: ", err)

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to process data"})

return

}

// Check if the proposal is already added in the map

if \_, exists := proposalsMap[proposalID]; !exists {

proposalsMap[proposalID] = &ProposalReport{

ProposalID: proposalID,

Title: title,

Description: proposalDescription,

Deadline: deadline,

ApprovedSubtasks: []SubtaskReport{},

}

}

// Add the approved subtask to the current proposal

subtask := SubtaskReport{

SubtaskID: subtaskID,

TopicName: topicName,

Description: subtaskDescription,

SkillsRequired: skillsRequired,

SubmissionDate: submissionDate,

Status: status,

ReviewerID: reviewerID,

ReviewStatus: reviewStatus,

}

proposalsMap[proposalID].ApprovedSubtasks = append(proposalsMap[proposalID].ApprovedSubtasks, subtask)

}

// Convert the proposals map to a slice

proposals := make([]ProposalReport, 0, len(proposalsMap))

for \_, proposal := range proposalsMap {

proposals = append(proposals, \*proposal)

}

c.JSON(http.StatusOK, proposals)

}

### 2. Register the New Route in main.go

We need to register the new route in the main.go file, passing the database connection to the handler:

go

Copy code

r.GET("/coordinator/proposals/:userId", func(c \*gin.Context) {

handlers.GetProposalsWithApprovedSubtasks(c, dbConn)

})

### 3. Update the ****main.go**** File

Ensure the main.go file imports the necessary packages and defines the new route:

go

Copy code

package main

import (

"log"

"github.com/gin-gonic/gin"

"github.com/gin-contrib/cors"

"database/sql"

\_ "github.com/go-sql-driver/mysql"

"your\_project\_name/db"

"your\_project\_name/migrations"

"your\_project\_name/handlers"

)

func main() {

dbConn, err := sql.Open("mysql", "root:root@tcp(127.0.0.1:3306)/hackathon")

if err != nil {

log.Fatal("Failed to connect to the database: ", err)

}

defer dbConn.Close()

err = migrations.RunMigrations(dbConn)

if err != nil {

log.Fatal("Migration failed: ", err)

}

r := gin.Default()

r.Use(cors.Default())

r.POST("/signup", func(c \*gin.Context) {

handlers.SignupHandler(c, dbConn)

})

r.POST("/login", func(c \*gin.Context) {

handlers.LoginHandler(c, dbConn)

})

r.POST("/proposal", func(c \*gin.Context) {

handlers.CreateProposalHandler(c, dbConn)

})

r.POST("/subtask", func(c \*gin.Context) {

handlers.CreateSubtaskHandler(c, dbConn)

})

r.POST("/subtask/upload", func(c \*gin.Context) {

handlers.UploadSubtaskHandler(c, dbConn)

})

r.POST("/subtask/review", func(c \*gin.Context) {

handlers.ReviewSubtaskHandler(c, dbConn)

})

// New route for coordinator's proposal and subtask tracking

r.GET("/coordinator/proposals/:userId", func(c \*gin.Context) {

handlers.GetProposalsWithApprovedSubtasks(c, dbConn)

})

r.Run(":8080")

}

### 4. curl Command to Test the New API

#### Endpoint: GET /coordinator/proposals/:userId

Replace 1 with the actual userId (coordinator ID) in the following curl command:

bash

Copy code

curl -X GET http://localhost:8080/coordinator/proposals/1

### Example Response:

json

Copy code

[

{

"proposal\_id": 1,

"title": "New Web Application",

"description": "A web app project for hackathon",

"deadline": "2024-12-31",

"approved\_subtasks": [

{

"subtask\_id": 2,

"topic\_name": "Frontend Development",

"description": "Develop the frontend using HTML, CSS, and JavaScript",

"skills\_required": "HTML, CSS, JavaScript",

"submission\_date": "2024-11-15",

"status": "completed",

"reviewer\_id": 1,

"review\_status": "approved"

}

]

}

]

### Explanation:

* **Query Logic**: We join the proposals, subtasks, and submissions tables to fetch proposals where the user is the coordinator, and get subtasks that have been approved.
* **Handler Logic**: The handler aggregates the data into a structure (ProposalReport) that includes both the proposal details and the approved subtasks.

This handler provides a reporting feature to track the progress of proposals and the status of subtasks for the coordinator. Let me know if you'd like to adjust anything!

## Prompt:

Can you suggest a structure for the README file for my proposal management system?

## Response:

Certainly! Here’s a suggested structure:  
```markdown  
# Proposal Management System  
  
## Overview  
A brief description of the project and its purpose.  
  
## Features  
- User authentication  
- Proposal creation and management  
- Subtask handling  
- File submissions and reviews  
- Notification system  
  
## Technologies  
- Golang  
- MySQL  
- Gin Framework  
- Bootstrap  
  
## Installation  
Instructions to set up the project locally.  
  
## Usage  
How to use the application once it's running.  
  
## License  
Information about the project's license.  
```

## Prompt:

I am getting below error while hitting this API from postmans: "File upload failed: http: no such file"

## Response:

The error you're encountering, "File upload failed: http: no such file", usually occurs when the API does not receive the file properly. This could happen for a few reasons, particularly around how the request is being made from Postman.

### Checklist for File Uploads in Postman:

1. **Ensure Correct HTTP Method**:
   * The request method should be POST.
2. **Correct multipart/form-data Setup**:
   * When uploading files, you must set the Content-Type to multipart/form-data.
   * In Postman, go to the **Body** tab, select **form-data**, and then:
     + Set the key to file (or the parameter expected by your backend).
     + For the value, use the **Select Files** button to pick the file from your system.
   * Include other fields like subtask\_id and user\_id as text form fields.
3. **Handler in Go to Accept Multipart Files**: The issue might also be related to how files are processed on the server. Based on the error message, let's revisit the handler to ensure everything is configured correctly.

### Corrected UploadSubtaskHandler

Here's an updated version of the UploadSubtaskHandler to handle file uploads properly:

go

Copy code

func UploadSubtaskHandler(c \*gin.Context) {

// Get form fields for subtask\_id and user\_id

subtaskIDStr := c.PostForm("subtask\_id")

userIDStr := c.PostForm("user\_id")

// Validate form data

if subtaskIDStr == "" || userIDStr == "" {

c.JSON(http.StatusBadRequest, gin.H{"error": "subtask\_id and user\_id are required"})

return

}

// Convert subtask\_id and user\_id from string to int

subtaskID, err := strconv.Atoi(subtaskIDStr)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid subtask\_id"})

return

}

userID, err := strconv.Atoi(userIDStr)

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "Invalid user\_id"})

return

}

// Retrieve the file from the form

file, err := c.FormFile("file")

if err != nil {

c.JSON(http.StatusBadRequest, gin.H{"error": "File upload failed: " + err.Error()})

return

}

// Save the uploaded file to a directory on the server

filePath := fmt.Sprintf("./uploads/%s", file.Filename)

if err := c.SaveUploadedFile(file, filePath); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Unable to save file: " + err.Error()})

return

}

// Create the submission record

submission := models.Submission{

SubtaskID: subtaskID,

UserID: userID,

FilePath: filePath,

SubmissionDate: time.Now(),

}

// Save submission in the database

if err := db.UploadSubtask(submission); err != nil {

c.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to save subtask submission"})

return

}

c.JSON(http.StatusOK, gin.H{"message": "Subtask uploaded successfully"})

}

### Key Points to Check:

1. **FormFile**: The file should be correctly passed from the request in the form-data body of the request, with the key set to "file".
2. **SaveUploadedFile**: This method saves the file to the specified path on the server. Ensure the path ("./uploads/") exists and has proper write permissions.

### Testing with Postman:

1. Set the request method to POST.
2. Use the URL http://localhost:8080/subtask/upload.
3. Go to the **Body** tab, choose **form-data**, and set the following key-value pairs:
   * subtask\_id (key) as 1 (value) (text)
   * user\_id (key) as 2 (value) (text)
   * file (key) and attach a file by selecting **Choose Files**.
4. Send the request.

### Curl Command for Testing:

Alternatively, use the following curl command to test the upload:

bash

Copy code

curl -X POST http://localhost:8080/subtask/upload \

-F "subtask\_id=1" \

-F "user\_id=2" \

-F "file=@/path/to/your/presentation.ppt"

### File Upload Directory:

Ensure that the ./uploads/ directory exists, or you can create it using:

bash

Copy code

mkdir -p ./uploads

Also, make sure that your Go process has write permissions to the ./uploads/ directory.

Let me know if the issue persists, or if you need further adjustments!

## Prompt:

Good job my campanion, backend is looking fine Now we need to have front end for this application. There KPMG blue and white in color Use bootstrap to make it screen compatible and I want you to be as advanced as possible for the UI There will be a login and signup page post login and signup user will be redirected to home screen Home Screen will be showing all available proposals from API (/proposals get method) Response of the API is below: [ { "id": 1, "title": "New Web Application", "description": "A web app project for hackathon", "coordinator\_id": 1, "deadline": "2024-09-30", "created\_at": "2024-09-27T21:50:32Z" } ] Make each proposal clickable which shows all subtasks of it from API (/proposals/1/subtasks get method) Response of the API is below: [ { "id": 1, "proposal\_id": 1, "topic\_name": "Frontend Development", "description": "Develop the frontend using HTML, CSS, and JavaScript", "skills\_required": "HTML, CSS, JavaScript", "submission\_date": "2024-09-29", "status": "pending" }, { "id": 2, "proposal\_id": 1, "topic\_name": "Backend Development", "description": "Develop the backend using Java, Python, and Mysql", "skills\_required": "Java, Python, Mysql", "submission\_date": "2024-09-29", "status": "pending" } ]

## Response:

Here’s a structured approach to build a responsive and visually appealing frontend using Bootstrap, following the KPMG blue and white color scheme.

### Project Structure

* **index.html**: The main HTML file for your application.
* **styles.css**: Custom styles (if needed).
* **scripts.js**: JavaScript for handling API calls.
* **Bootstrap**: Use CDN links for Bootstrap.

### 1. HTML Structure (index.html)

html

Copy code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Proposal Management</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">

<link rel="stylesheet" href="styles.css">

</head>

<body class="bg-light">

<nav class="navbar navbar-expand-lg navbar-light bg-white">

<a class="navbar-brand" href="#">Proposal Management</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

</nav>

<div class="container mt-4">

<div id="login-section" class="col-md-6 mx-auto">

<h2>Login</h2>

<div class="form-group">

<input type="text" class="form-control" id="login-username" placeholder="Username">

</div>

<div class="form-group">

<input type="password" class="form-control" id="login-password" placeholder="Password">

</div>

<button class="btn btn-primary" onclick="login()">Login</button>

<button class="btn btn-secondary" onclick="showSignup()">Signup</button>

</div>

<div id="signup-section" class="col-md-6 mx-auto" style="display: none;">

<h2>Signup</h2>

<div class="form-group">

<input type="text" class="form-control" id="signup-username" placeholder="Username">

</div>

<div class="form-group">

<input type="password" class="form-control" id="signup-password" placeholder="Password">

</div>

<div class="form-group">

<input type="text" class="form-control" id="signup-name" placeholder="Name">

</div>

<div class="form-group">

<input type="text" class="form-control" id="signup-designation" placeholder="Designation">

</div>

<button class="btn btn-primary" onclick="signup()">Signup</button>

<button class="btn btn-secondary" onclick="showLogin()">Back to Login</button>

</div>

<div id="home-section" style="display: none;">

<h2>Available Proposals</h2>

<div id="proposals-list" class="list-group"></div>

</div>

<div id="subtasks-section" class="mt-4" style="display: none;">

<h3>Subtasks for <span id="proposal-title"></span></h3>

<div id="subtasks-list" class="list-group"></div>

<button class="btn btn-secondary mt-3" onclick="goBack()">Back to Proposals</button>

</div>

</div>

<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js"></script>

<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.9.2/dist/umd/popper.min.js"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js"></script>

<script src="scripts.js"></script>

</body>

</html>

### 2. CSS Styles (styles.css)

To apply the KPMG blue color scheme, you can customize Bootstrap colors or add additional styles.

css

Copy code

body {

background-color: #f7f7f7;

}

.navbar {

background-color: #005EB8; /\* KPMG blue \*/

color: white;

}

.navbar-brand {

color: white !important;

}

.btn-primary {

background-color: #005EB8; /\* KPMG blue \*/

border-color: #005EB8;

}

.btn-primary:hover {

background-color: #004A8D; /\* Darker shade \*/

border-color: #004A8D;

}

.list-group-item {

cursor: pointer;

}

### 3. JavaScript Logic (scripts.js)

Implement the login, signup, and proposal fetching functionalities.

javascript

Copy code

function showSignup() {

document.getElementById('login-section').style.display = 'none';

document.getElementById('signup-section').style.display = 'block';

}

function showLogin() {

document.getElementById('signup-section').style.display = 'none';

document.getElementById('login-section').style.display = 'block';

}

function login() {

const username = document.getElementById('login-username').value;

const password = document.getElementById('login-password').value;

fetch('http://localhost:8080/login', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ username, password })

})

.then(response => response.json())

.then(data => {

if (data.success) {

document.getElementById('login-section').style.display = 'none';

document.getElementById('home-section').style.display = 'block';

fetchProposals();

} else {

alert('Login failed: ' + data.error);

}

})

.catch(error => console.error('Error:', error));

}

function signup() {

const username = document.getElementById('signup-username').value;

const password = document.getElementById('signup-password').value;

const name = document.getElementById('signup-name').value;

const designation = document.getElementById('signup-designation').value;

fetch('http://localhost:8080/signup', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ username, password, name, designation })

})

.then(response => response.json())

.then(data => {

if (data.success) {

alert('Signup successful! You can now log in.');

showLogin();

} else {

alert('Signup failed: ' + data.error);

}

})

.catch(error => console.error('Error:', error));

}

function fetchProposals() {

fetch('http://localhost:8080/proposals')

.then(response => response.json())

.then(data => {

const proposalsList = document.getElementById('proposals-list');

proposalsList.innerHTML = ''; // Clear previous proposals

data.forEach(proposal => {

const div = document.createElement('a');

div.className = 'list-group-item list-group-item-action';

div.innerHTML = `<strong>${proposal.title}</strong> - ${proposal.description}`;

div.onclick = () => fetchSubtasks(proposal.id, proposal.title);

proposalsList.appendChild(div);

});

})

.catch(error => console.error('Error:', error));

}

function fetchSubtasks(proposalID, title) {

document.getElementById('home-section').style.display = 'none';

document.getElementById('subtasks-section').style.display = 'block';

document.getElementById('proposal-title').innerText = title;

fetch(`http://localhost:8080/proposals/${proposalID}/subtasks`)

.then(response => response.json())

.then(data => {

const subtasksList = document.getElementById('subtasks-list');

subtasksList.innerHTML = ''; // Clear previous subtasks

data.forEach(subtask => {

const div = document.createElement('div');

div.className = 'list-group-item';

div.innerHTML = `<strong>${subtask.topic\_name}</strong> - ${subtask.description} (Due: ${subtask.submission\_date})`;

subtasksList.appendChild(div);

});

})

.catch(error => console.error('Error:', error));

}

function goBack() {

document.getElementById('subtasks-section').style.display = 'none';

document.getElementById('home-section').style.display = 'block';

}

### Summary

* **Responsive Design**: The application uses Bootstrap to ensure it’s responsive and looks good on different screen sizes.
* **KPMG Blue Theme**: The color scheme is based on KPMG’s branding.
* **Functionality**: The application allows users to log in, sign up, view proposals, and see subtasks related to each proposal.

### Next Steps

1. **Run Your Backend**: Ensure your backend APIs are running and accessible.
2. **Test the Frontend**: Open index.html in a web browser and test the application’s functionality.
3. **Additional Features**: You can extend the functionality by adding more features such as file uploads, submission forms, or notifications.

Feel free to reach out if you need any more enhancements or help with specific functionalities!

## Prompt:

my login api is returning 200 OK with below response: { "message": "Login successful", "user\_id": 2 } But the UI is showing Login failed: undefined Also store the user\_id somewhere on the fly as it will be required for other API calls

## Response:

To resolve the issue with the UI showing "Login failed: undefined," you need to update your JavaScript code to correctly handle the successful response from the login API. Additionally, you should store the user\_id for later API calls.

Here's how you can modify the login function and include a way to store the user\_id:

### Updated JavaScript Function (scripts.js)

javascript

Copy code

let userId = null; // Variable to store user\_id

function login() {

const email = document.getElementById('login-email').value;

const password = document.getElementById('login-password').value;

fetch('http://localhost:8080/login', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ email, password })

})

.then(response => {

if (!response.ok) {

throw new Error('Network response was not ok');

}

return response.json();

})

.then(data => {

if (data.message === "Login successful") {

userId = data.user\_id; // Store user\_id for later use

document.getElementById('login-section').style.display = 'none';

document.getElementById('home-section').style.display = 'block';

fetchProposals();

} else {

alert('Login failed: ' + data.error || 'Invalid credentials');

}

})

.catch(error => {

console.error('Error:', error);

alert('Login failed: ' + error.message); // Display network error

});

}

### Key Changes

1. **Storing user\_id**: Added a global variable userId to store the logged-in user's ID.
2. **Error Handling**: Improved error handling for the fetch request, including a check for the response.ok status.
3. **Response Check**: Updated the check for a successful login to look for data.message instead of a success property.

### Summary

With these changes:

* If the login is successful, the user\_id will be stored in the userId variable, which you can use later for other API calls.
* If there's a network error or the login fails, the UI will show an appropriate message.

### Testing

1. **Reload your HTML file** in the browser after saving the changes.
2. **Test the login form** to ensure that it correctly processes both successful and failed login attempts.

Feel free to let me know if you have any more questions or need further modifications!

## Prompt:

Write an db function that updates the review status in submissions based on the submissionId And if it is approved then we need to update the status for subtask as well to completed

## Response:

Here is a MySQL function that updates the review status in the submissions table based on submission\_id and, if the status is approved, also updates the status for the related subtask to "completed".

sql

Copy code

DELIMITER //

CREATE PROCEDURE UpdateReviewStatus(

IN p\_submission\_id INT,

IN p\_review\_status VARCHAR(20)

)

BEGIN

-- Update the review status for the specified submission

UPDATE submissions

SET review\_status = p\_review\_status

WHERE submission\_id = p\_submission\_id;

-- If the review status is approved, update the subtask status to 'completed'

IF p\_review\_status = 'approved' THEN

DECLARE subtask\_id INT;

-- Get the subtask ID associated with the submission

SELECT subtask\_id INTO subtask\_id

FROM submissions

WHERE submission\_id = p\_submission\_id;

-- Update the subtask status to 'completed'

UPDATE subtasks

SET status = 'completed'

WHERE id = subtask\_id;

END IF;

END//

DELIMITER ;

### Explanation

1. **Procedure Name**: The procedure is called UpdateReviewStatus.
2. **Parameters**:
   * p\_submission\_id: The ID of the submission whose review status you want to update.
   * p\_review\_status: The new review status (e.g., "approved" or "rejected").
3. **Update Review Status**: The first UPDATE query updates the review\_status in the submissions table based on submission\_id.
4. **Update Subtask Status if Approved**:
   * If the new review status is "approved," the procedure finds the corresponding subtask\_id and updates its status to "completed."

To call this procedure, use:

sql

Copy code

CALL UpdateReviewStatus(1, 'approved');

This call would update the submission with ID 1 to be "approved" and, if applicable, update the related subtask to "completed.

## Prompt:

Can you suggest a structure for the README file for my proposal management system?

## Response:

Here’s a sample README file for your proposal management system project:

markdown

Copy code

# Proposal Management System

## Overview

The Proposal Management System is a web application that facilitates the management of proposals, subtasks, and submissions. It allows users to create proposals, manage subtasks, submit files, and review submissions efficiently. The application is built using the Gin framework for the backend with a MySQL database and a simple frontend using Bootstrap, HTML, CSS, and JavaScript.

## Features

- User authentication (Signup and Login)

- Proposal creation and management

- Subtask management for each proposal

- Submission of files for specific subtasks

- Review submissions with approval/rejection functionality

- Notifications for new proposals, subtasks, and submission reviews

## Tech Stack

- \*\*Backend\*\*: Golang (Gin framework)

- \*\*Database\*\*: MySQL

- \*\*Frontend\*\*: HTML, CSS (Bootstrap), JavaScript (AJAX)

- \*\*Cloud Services\*\*: Azure Cloud Services (if applicable)

## Getting Started

### Prerequisites

- Go (1.16 or later)

- MySQL server

- Git (optional, for version control)

### Installation

1. Clone the repository:

```bash

git clone <repository-url>

cd proposal-management-system

1. Set up the MySQL database:
   * Create a MySQL database for the application.
   * Run the provided SQL scripts to create the required tables.
2. Configure the database connection in the code (if needed):
   * Update the connection settings in your Golang code to point to your MySQL database.
3. Run the application:

bash

Copy code

go run main.go

1. Open your web browser and navigate to:

arduino

Copy code

http://localhost:8080

## API Endpoints

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /signup | POST | Register a new user |
| /login | POST | Authenticate user |
| /proposals | GET | Retrieve all proposals |
| /subtasks | GET | Retrieve all subtasks for a proposal |
| /submissions | POST | Submit a file for a specific subtask |
| /notifications | GET | Retrieve notifications for a user |

## Database Schema

The application uses the following database tables:

* **users**: Stores user information including name, email, and password.
* **proposals**: Contains proposal details and is linked to users (coordinators).
* **subtasks**: Holds information about subtasks for each proposal.
* **submissions**: Stores submissions made by users for specific subtasks.
* **notifications**: Keeps track of notifications related to proposals, subtasks, and submissions.

## Contributing

Contributions are welcome! If you have suggestions for improvements or find any issues, please open an issue or submit a pull request.

## License

This project is licensed under the MIT License. See the LICENSE file for details.

## Acknowledgments

* [Gin Gonic](https://github.com/gin-gonic/gin) for the web framework
* [MySQL](https://www.mysql.com/) for the database
* [Bootstrap](https://getbootstrap.com/) for frontend styling

vbnet

Copy code

### Instructions for Customization

- Replace `<repository-url>` with the actual URL of your project repository.

- You may want to adjust the installation and configuration instructions based on your project's specifics.

- Add any additional features or acknowledgments relevant to your project.

Feel free to let me know if you need any modifications or additional sections!