ROUEN / ZONE01

Forum Project

AND ALL OPTIONAL

Arnaud, Yanis, Thomas, Rafta & Lucas

OCTOBER 7, 2024

Presentation of project architecture and processes.

Summary

Git is an essential workflow for efficient collaboration and version control on Gitea. It provides a structured approach to managing your project's development lifecycle.

Gitflow

GITEA DEMONSTRATION AND PRACTICE

Master Branch: This is the main branch containing stable, production-ready code. It should always reflect the latest released version.

Feature Branches: Create separate branches for each new feature or task. This allows developers to work independently without affecting the main codebase.

Issues: Use issues to track tasks, bugs, and feature requests. They help organize work and facilitate communication within the team.

Linking to Issues: When making commits and PR, reference the related issue number (e.g., "#10 Bug Fixed"). This creates a clear connection between code changes and project tasks.

Pull Requests

NEVER MERGE AUTOMATICALLY

Create Pull Request: Go back to Gitea, navigate to your branch, click on "New Pull Request", name the pull request, referencing the issue number (e.g., "#42 Feature: Add login functionality").

Code Review: Wait for a team member to review your code, address any feedback or changes requested.

Merge: once approved, the pull request can be merged into the master branch

Summary

Here is an outline of the features required for the forum project.

Technical Breakdown

FEATURES SUMMARY

- Database
  - Database DBML, Setup, Connections, CRUD functions, Security & Encryption
- Authentification
  - Login
  - Register
  - Cookies (UUID)
  - User Roles
- Communication
  - Posts
  - Comments
  - Likes/Dislikes
  - Filters
  - Activity
- Security
  - HTTPS && SSL certificate
  - Go TLS structure configuration

Here is an outline of the authentification features for the forum project.

Authentification (1/2)

AUTHENTIFICATION SUMMARY

- **Login:**
  - Must ask for email (return an error if the email is already taken)
  - Must ask for username
  - Must ask for password (password need to be encrypted)
- **Register:**
  - User should be able to register and login using Google and GitHub
- **Cookies (UUID)**

Here is an outline of the authentification features for the forum project.

Authentification (2/2)

AUTHENTIFICATION SUMMARY

- **User Roles:**
  - We should implement at least 4 types of users:
    - Guests: Can't interact with the forum, only see the content
    - Users: Able to create, comment, like/dislike posts, but their posts are not instantly visible and should be validated by a moderator
    - Moderators: Should be able to delete or report posts to the admins. The report should be attached with these categories (irrelevant, obscene, illegal or insulting). Users can become moderators by requesting it, admins should be able to accept or not that role
    - Administrators: Should be able to change the role of any users. Receive reports from moderators and be able to respond. Be able to respond to moderators role requests from users. Administrators should also be able to delete posts and comments. But also be able to create and delete categories

Here is an outline of the communication features for the forum project.

Communication (1/3)

COMMUNICATION SUMMARY

- **Posts:**
  - Only registered users can post
  - You can't post an empty post.
  - User need to select a category where to create the post, several categories can be selected.
  - Posts should be visible for all users
  - User should be able to upload an image into the post with at least .jpg, .png & .gif (the max size of the img should be 20mb. if it's superior then return an error)
  - User should be able to edit/remove their own posts, likes/dislikes and comments

Here is an outline of the communication features for the forum project.

Communication (2/3)

COMMUNICATION SUMMARY

- **Comments:**
  - Only registered users can comment
  - Comments should be visible for all users
- **Likes/Dislikes:**
  - You can't like and dislike something at the same time.
  - Posts and comments can be liked and disliked
  - Only registered users can like/dislike
  - The number of likes should be visible for all users

Here is an outline of the communication features for the forum project.

Communication (3/3)

COMMUNICATION SUMMARY

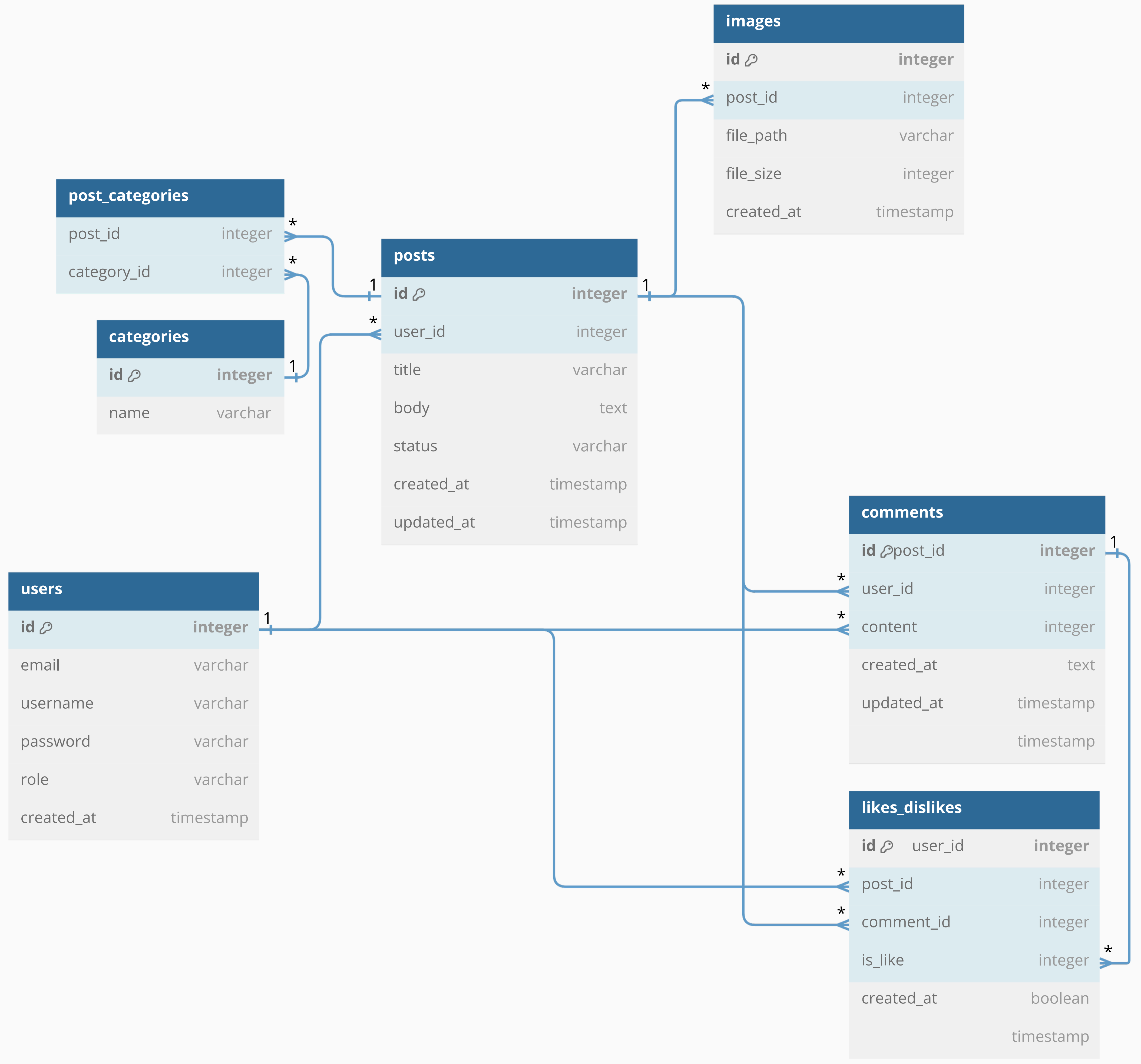
- **Filters:**
  - Categories
  - Last created
  - Most liked
- **Activity:**
  - Users should be notified when their posts are liked/disliked or commented
  - Create an activity page where the user can monitor their posts, their likes/dislikes, where and what the user has been commenting

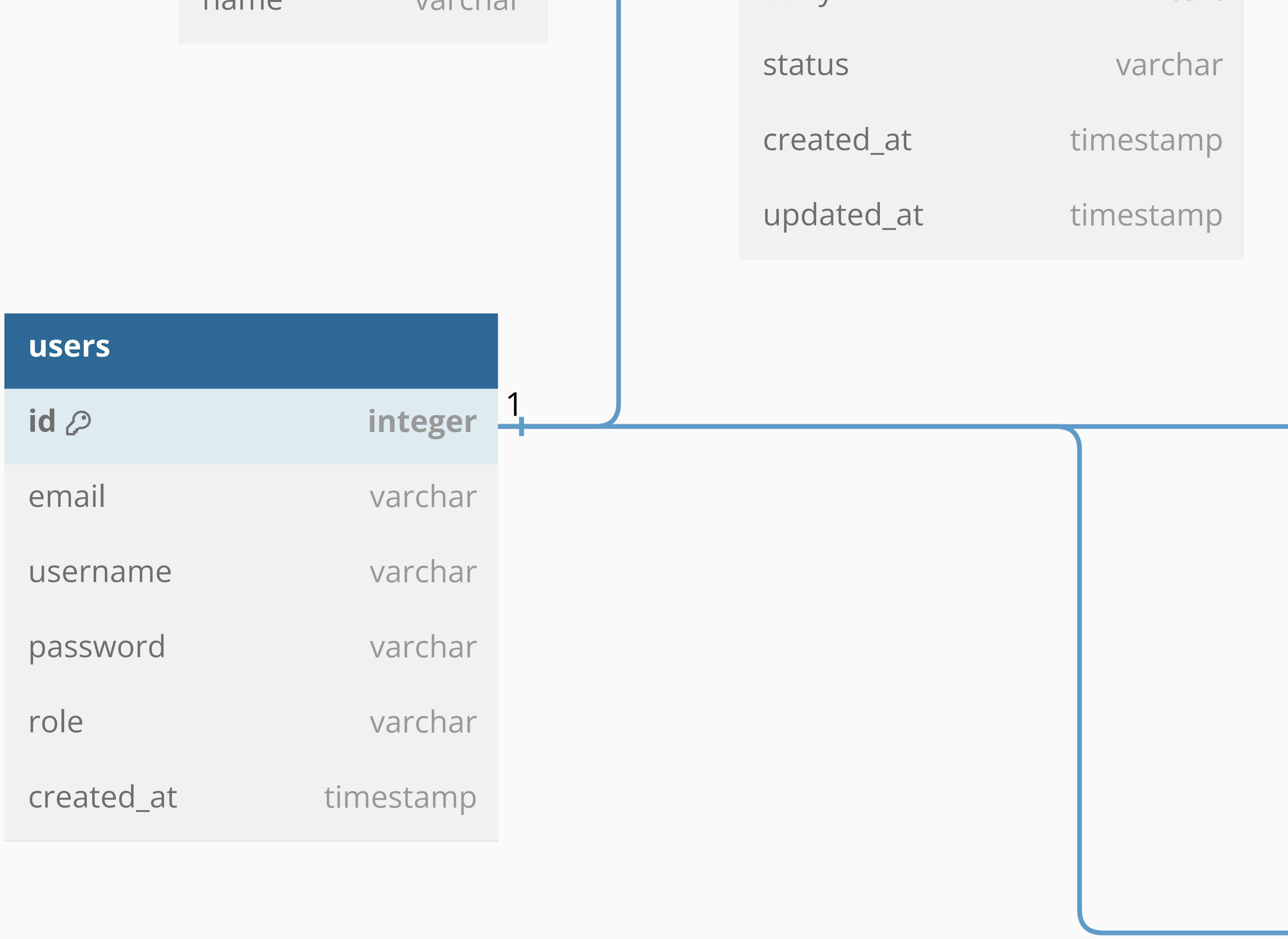
Here is an outline of the security features for the forum project.

Security

SECURITY SUMMARY

- **Go TLS structure configuration:**
  - Implement a Hypertext Transfer Protocol Secure (HTTPS) (generate an SSL certificate). (cf. autocert package go, openssl manual)
  - A way to configure the certificates information, either via .env, config files or another method
- **Server Security:**
  - Rate limiting
  - Handle 404, 400 & 500 errors.
  - Server Timeout reduced.
- **Encryption:**
  - Database should be encrypted and have a password to access it.
  - Client session cookies should be unique, the session state is stored on the server and the session should present a unique identifier. (cf. UUID, bcrypt)

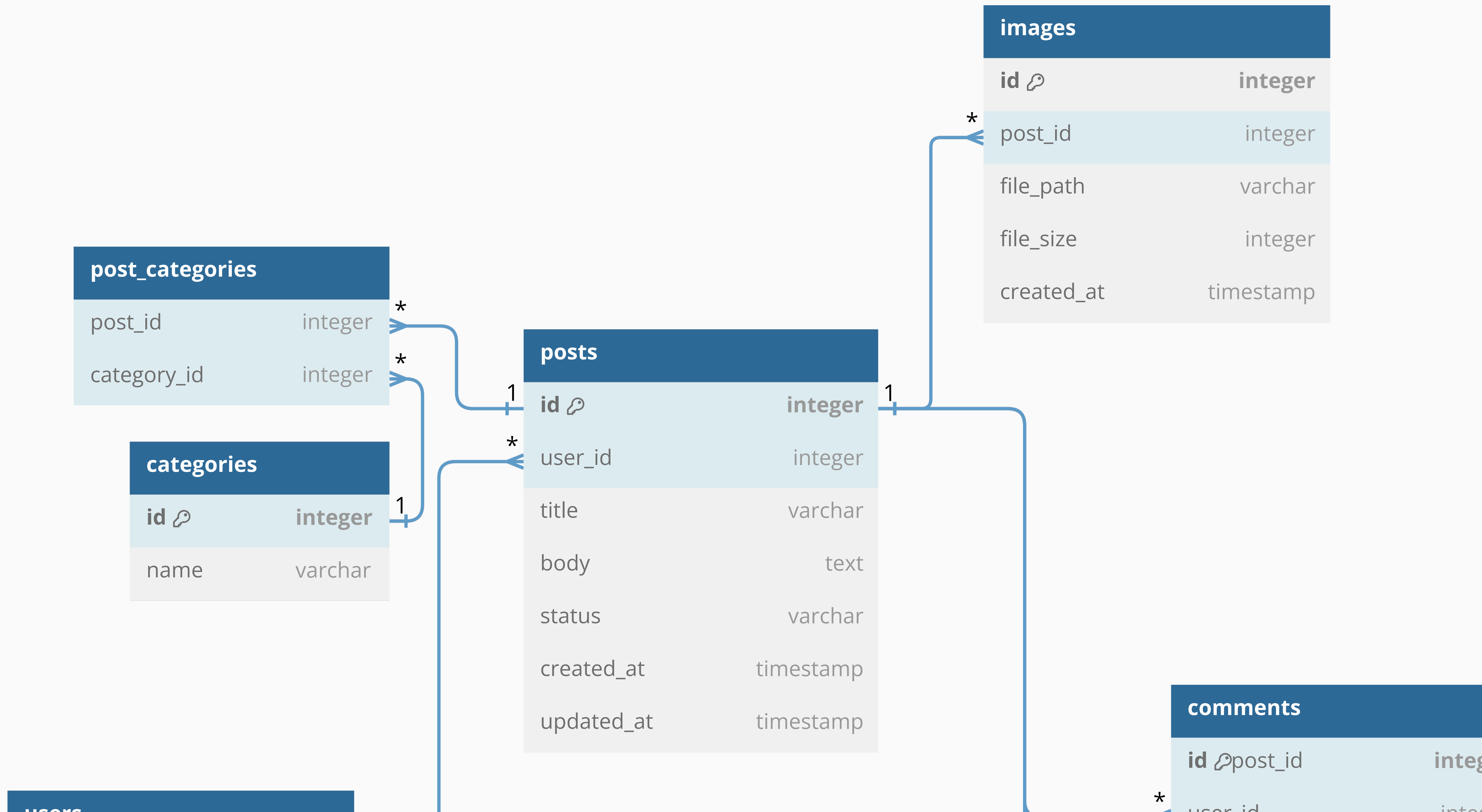

images
id integer
post_id integer
file_path varchar
file_size integer
created_at timestamp
post_categories
post_id integer
category_id integer
categories
id integer
name varchar
posts
id integer
user_id integer
title varchar
body text
status varchar
created_at timestamp
updated_at timestamp
comments
id post_id integer
user_id integer
content integer
created_at text
updated_at timestamp
timestamp
users
id integer
email varchar
username varchar
password varchar
role varchar
created_at timestamp
likes_dislikes
id user_id integer
post_id integer
comment_id integer
is_like integer
created_at boolean
timestamp
1
*

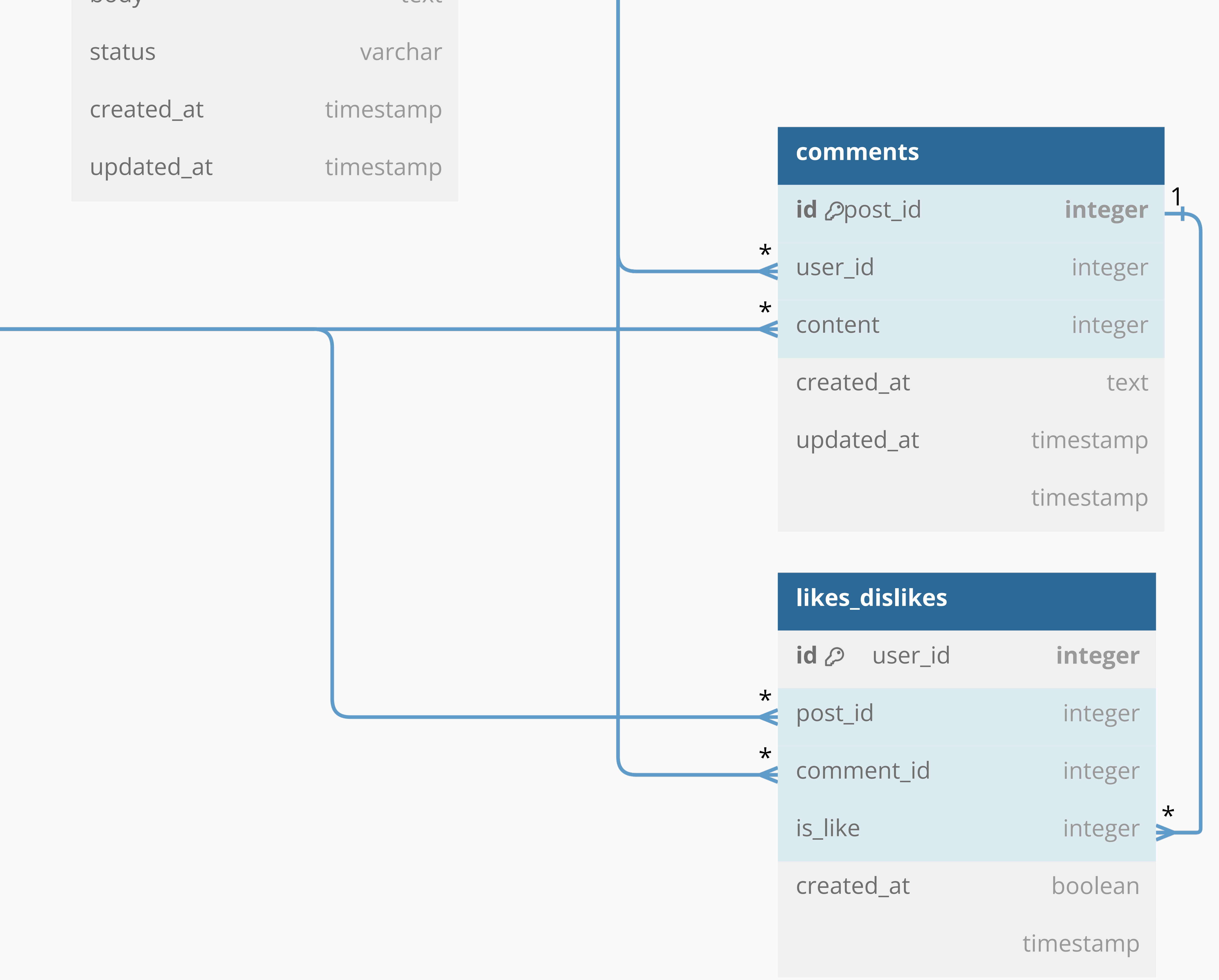


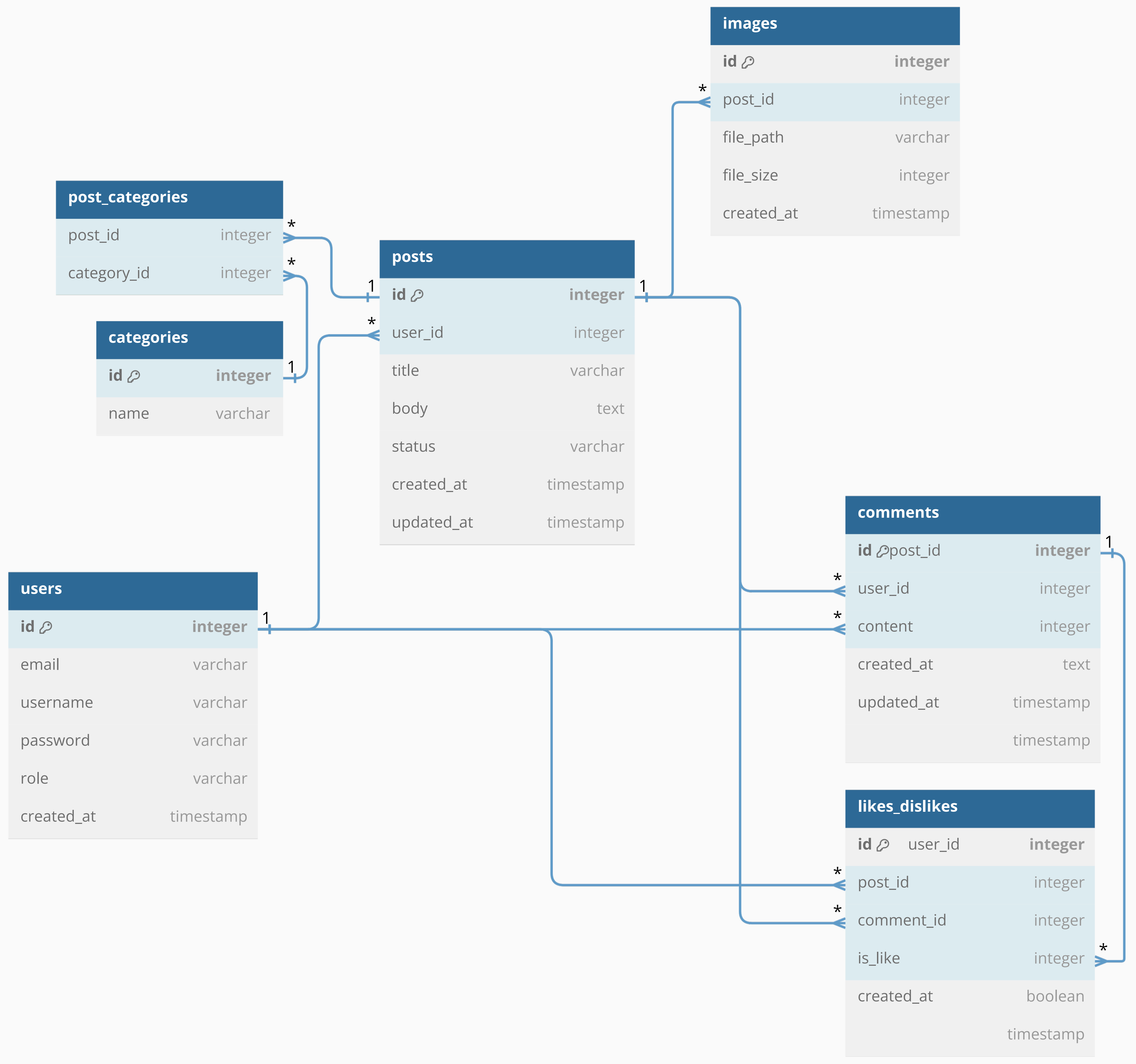
name	varchar

status	varchar
created_at	timestamp
updated_at	timestamp

users	
id	**integer**
email	varchar
username	varchar
password	varchar
role	varchar
created_at	timestamp


images
id integer
post_id integer
file_path varchar
file_size integer
created_at timestamp
post_categories
post_id integer
category_id integer
categories
id integer
name varchar
posts
id integer
user_id integer
title varchar
body text
status varchar
created_at timestamp
updated_at timestamp
comments
id post_id
*
1
*
*
1
*
1
*


status varchar
created_at timestamp
updated_at timestamp
comments
id post_id integer
user_id integer
content integer
created_at text
updated_at timestamp
timestamp
likes_dislikes
id user_id integer
post_id integer
comment_id integer
is_like integer
created_at boolean
timestamp
1
*
*
*
*
*


images
id integer
post_id integer
file_path varchar
file_size integer
created_at timestamp
post_categories
post_id integer
category_id integer
categories
id integer
name varchar
posts
id integer
user_id integer
title varchar
body text
status varchar
created_at timestamp
updated_at timestamp
comments
id post_id integer
user_id integer
content integer
created_at text
updated_at timestamp
timestamp
users
id integer
email varchar
username varchar
password varchar
role varchar
created_at timestamp
likes_dislikes
id user_id integer
post_id integer
comment_id integer
is_like integer
created_at boolean
timestamp
1
*

Summary

Day-by-day Tasks

TASK BREAKDOWN APPROACH

- **Overview:**
  - Project divided into 2 weeks
  - Daily task allocation for focused development
  - Logical progression of features
- **Key Principles:**
  - Start with core infrastructure
  - Build authentication early
  - Develop features incrementally
  - Address security throughout
  - End with optimization and deployment

Day-by-day Tasks

WEEKLY FOCUS

- **Week 1:**
  - Days 1-2: Project Setup and Database
  - Days 3-4: Authentication System
  - Days 5-6: User Management
  - Day 7: Post Management
- **Week 2:**
  - Days 8-9: Comment System and Like/Dislike
  - Day 10: Filters, Sorting, and Categories
  - Day 11: Activity and Notifications
  - Day 12: Moderation
  - Day 13: Security
  - Day 14: Final Touches and Docker

Summary

Here is an outline of the forum project's file structure.

Project Tree Structure

```
.
├── cmd
│   └── golang-server-layout
├── config
├── docs
├── internal
│   ├── database
│   ├── handlers
│   ├── middlewares
│   └── server
├── static
│   ├── assets
│   │   └── img
│   └── css
├── tests
└── web
    ├── pages
    └── templates
```

Here is an outline of the forum project's file structure.

Project Tree Structure

```
.
├──── cmd              ← cmd: Entry point for the application
│     └──── golang-server-layout
├──── config             ← config: Configuration management
├──── docs             ← docs: Project documentation
├──── internal           ← internal: Core application logic
│     ├──── database
│     ├──── handlers
│     ├──── middlewares
│     └──── server
├──── static           ← static: Static assets (CSS, images)
│     ├──── assets
│     │     └──── img
│     └──── css
├──── tests            ← tests: Test files
└──── web              ← web: HTML templates and pages
      ├──── pages
      └──── templates
```

Here is an outline of the forum project's file structure.

Internal Structure Deep Dive

```
.
├──── database
│    ├──── create_user.go
│    ├──── database.db
│    ├──── db.go
│    ├──── db_models.go
│    ├──── get_users.go
├──── handlers
│    ├──── handlers_models.go
│    └──── index_handler.go
├──── middlewares
│    ├──── cors.go
│    ├──── go.mod
│    ├──── rate_limiting.go
└──── server
 ├──── server.go
 └──── server_model.go
```

Here is an outline of the forum project's file structure.

Internal Structure Deep Dive

```
.
├──── database  ← database: Database operations and models
│   ├──── create_user.go      (CRUD operations for users)
│   ├──── database.db         (Database connection management)
│   ├──── db.go
│   ├──── db_models.go
│   ├──── get_users.go
├──── handlers  ← handlers: HTTP request handlers
│   ├──── handlers_models.go      (Specific handlers for different routes)
│   └──── index_handler.go
├──── middlewares   ← middlewares: Request processing
│   ├──── cors.go                (middlewaresAuthentication, CORS, logging, etc.)
│   ├──── go.mod
│   ├──── rate_limiting.go
└──── server  ← server: Server setup and configuration
 ├──── server.go
 └──── server_model.go
```

Summary

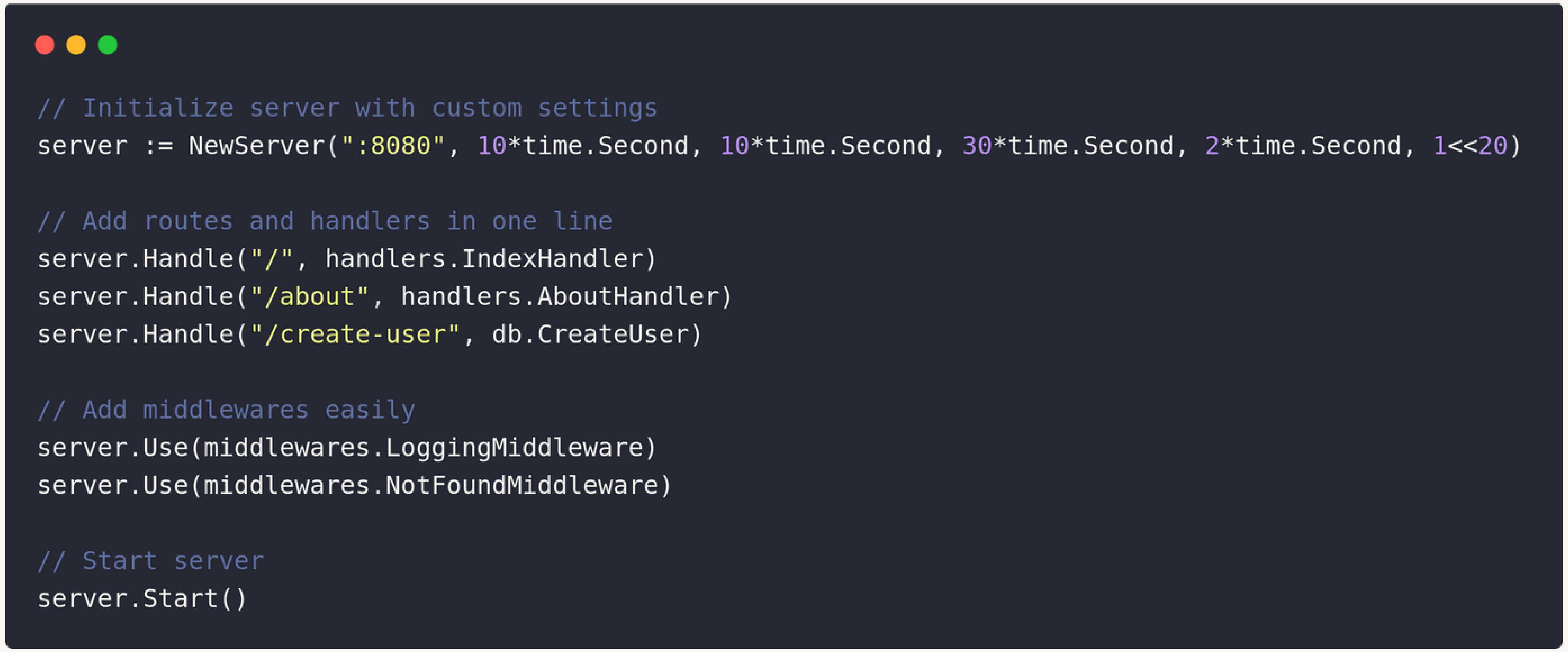
Simplified Server Configuration

Benefits:

- Streamlined server setup process
- Improved code readability and maintainability
- Flexible configuration for timeouts and settings
- Easy integration of custom handlers and middlewares

Server Wrapper Overview:

- Custom server initialization with one-line setup
- Easy route and handler association
- Simple middleware integration

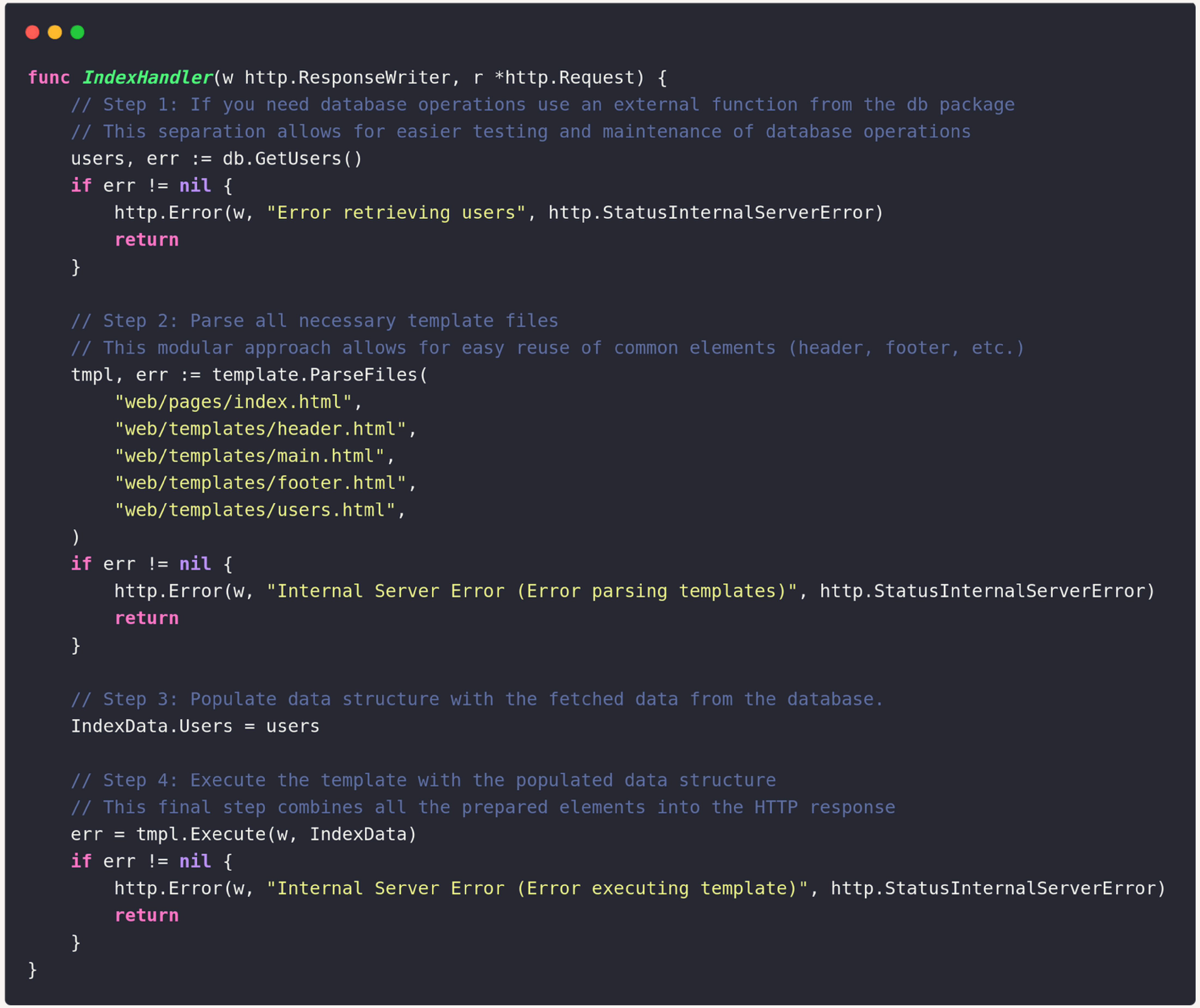


```go
// Initialize server with custom settings
server := NewServer(":8080", 10*time.Second, 10*time.Second, 30*time.Second, 2*time.Second, 1<<20)

// Add routes and handlers in one line
server.Handle("/", handlers.IndexHandler)
server.Handle("/about", handlers.AboutHandler)
server.Handle("/create-user", db.CreateUser)

// Add middlewares easily
server.Use(middlewares.LoggingMiddleware)
server.Use(middlewares.NotFoundMiddleware)

// Start server
server.Start()
```

Handler Architecture

Benefits:

- Clean and maintainable code structure
- Easy to add new routes and handlers
- Consistent data flow across different handlers

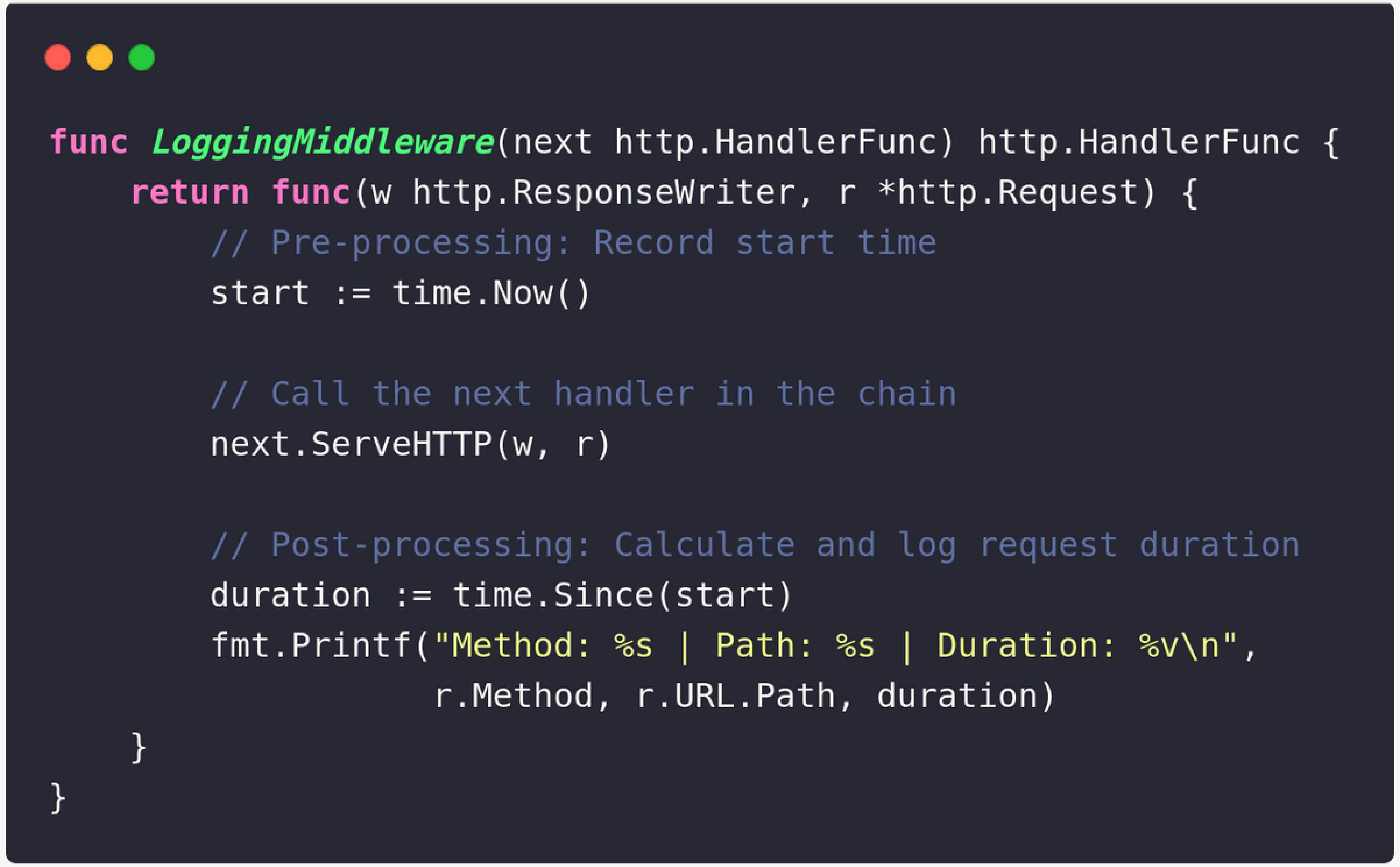


```go
func IndexHandler(w http.ResponseWriter, r *http.Request) {
    // Step 1: If you need database operations use an external function from the db package
    // This separation allows for easier testing and maintenance of database operations
    users, err := db.GetUsers()
    if err != nil {
        http.Error(w, "Error retrieving users", http.StatusInternalServerError)
        return
    }

    // Step 2: Parse all necessary template files
    // This modular approach allows for easy reuse of common elements (header, footer, etc.)
    tmpl, err := template.ParseFiles(
        "web/pages/index.html",
        "web/templates/header.html",
        "web/templates/main.html",
        "web/templates/footer.html",
        "web/templates/users.html",
    )
    if err != nil {
        http.Error(w, "Internal Server Error (Error parsing templates)", http.StatusInternalServerError)
        return
    }

    // Step 3: Populate data structure with the fetched data from the database.
    IndexData.Users = users

    // Step 4: Execute the template with the populated data structure
    // This final step combines all the prepared elements into the HTTP response
    err = tmpl.Execute(w, IndexData)
    if err != nil {
        http.Error(w, "Internal Server Error (Error executing template)", http.StatusInternalServerError)
        return
    }
}
```

Middleware Architecture

Middleware Overview:

- Modular approach to request/response processing
- Easily chainable and reusable across different routes



```go
func LoggingMiddleware(next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // Pre-processing: Record start time
        start := time.Now()

        // Call the next handler in the chain
        next.ServeHTTP(w, r)

        // Post-processing: Calculate and log request duration
        duration := time.Since(start)
        fmt.Printf("Method: %s | Path: %s | Duration: %v\n",
                   r.Method, r.URL.Path, duration)
    }
}
```

Benefits:

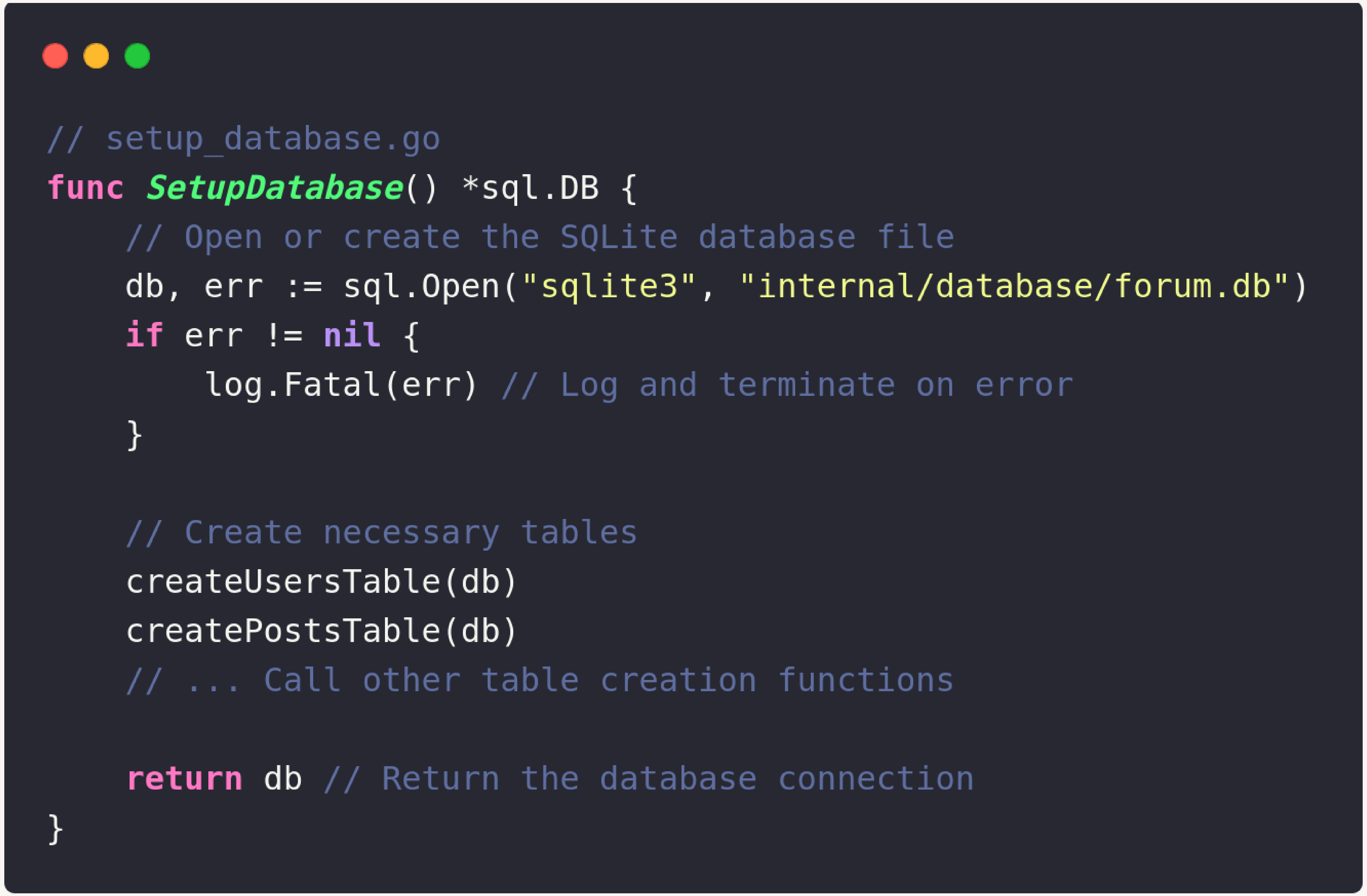
- Reusability: Can be applied to multiple routes or globally
- Easy Integration: Simple to add to the server setup
- Extensibility: Easy to add more middlewares for different purposes (e.g., authentication, CORS)

Summary

Database Setup

Setup Process:

1. **Define SetupDatabase() function to initialize the database.**
2. Create table creation functions for each entity.
3. Implement a utility function for executing SQL statements.

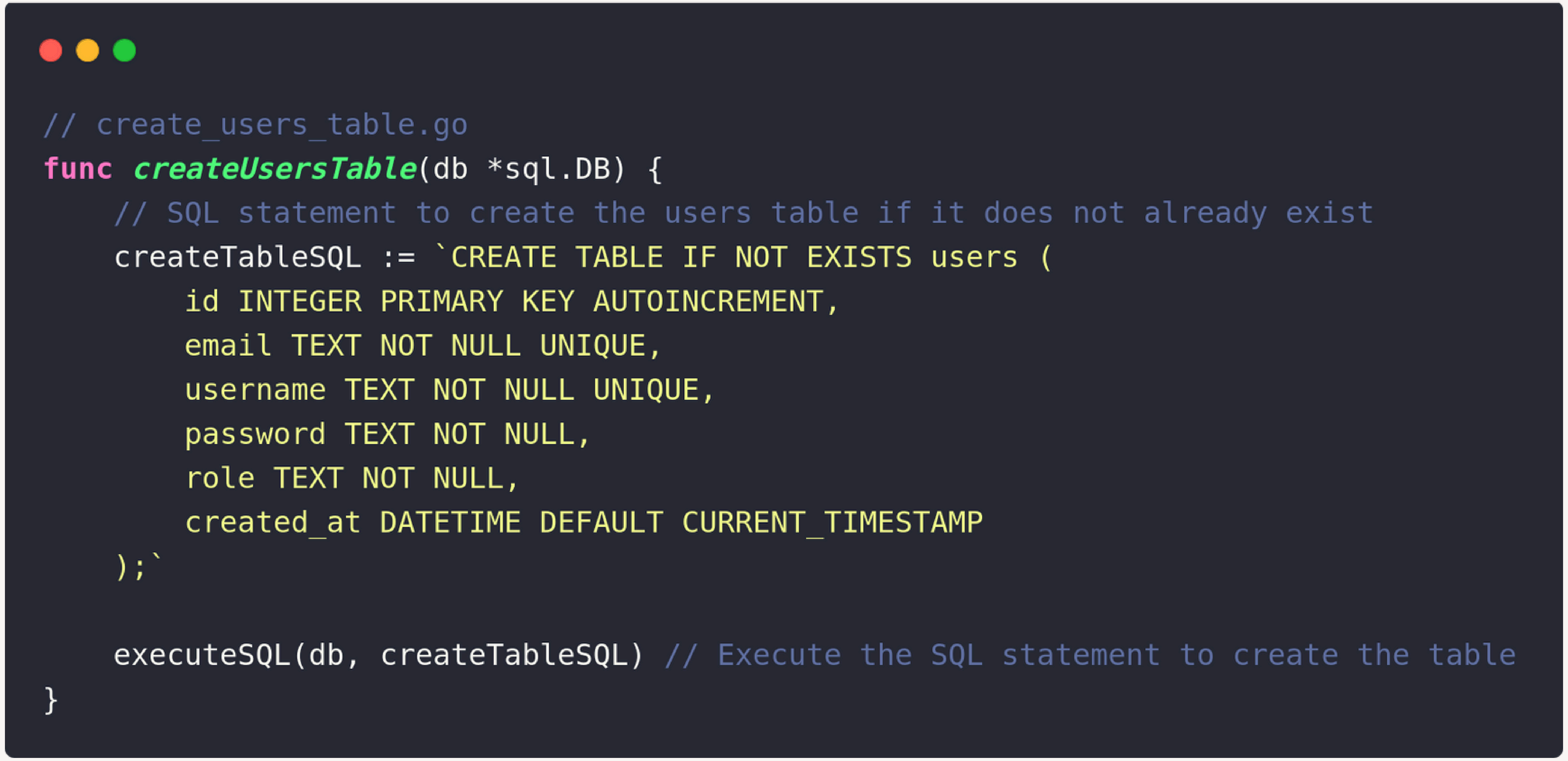


```go
// setup_database.go
func SetupDatabase() *sql.DB {
    // Open or create the SQLite database file
    db, err := sql.Open("sqlite3", "internal/database/forum.db")
    if err != nil {
        log.Fatal(err) // Log and terminate on error
    }

    // Create necessary tables
    createUsersTable(db)
    createPostsTable(db)
    // ... Call other table creation functions

    return db // Return the database connection
}
```

Database Table Creation

Setup Process:

1. Define SetupDatabase() function to initialize the database.
2. **Create table creation functions for each entity.**
3. Implement a utility function for executing SQL statements.



```go
// create_users_table.go
func createUsersTable(db *sql.DB) {
    // SQL statement to create the users table if it does not already exist
    createTableSQL := `CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        email TEXT NOT NULL UNIQUE,
        username TEXT NOT NULL UNIQUE,
        password TEXT NOT NULL,
        role TEXT NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP
    );`

    executeSQL(db, createTableSQL) // Execute the SQL statement to create the table
}
```

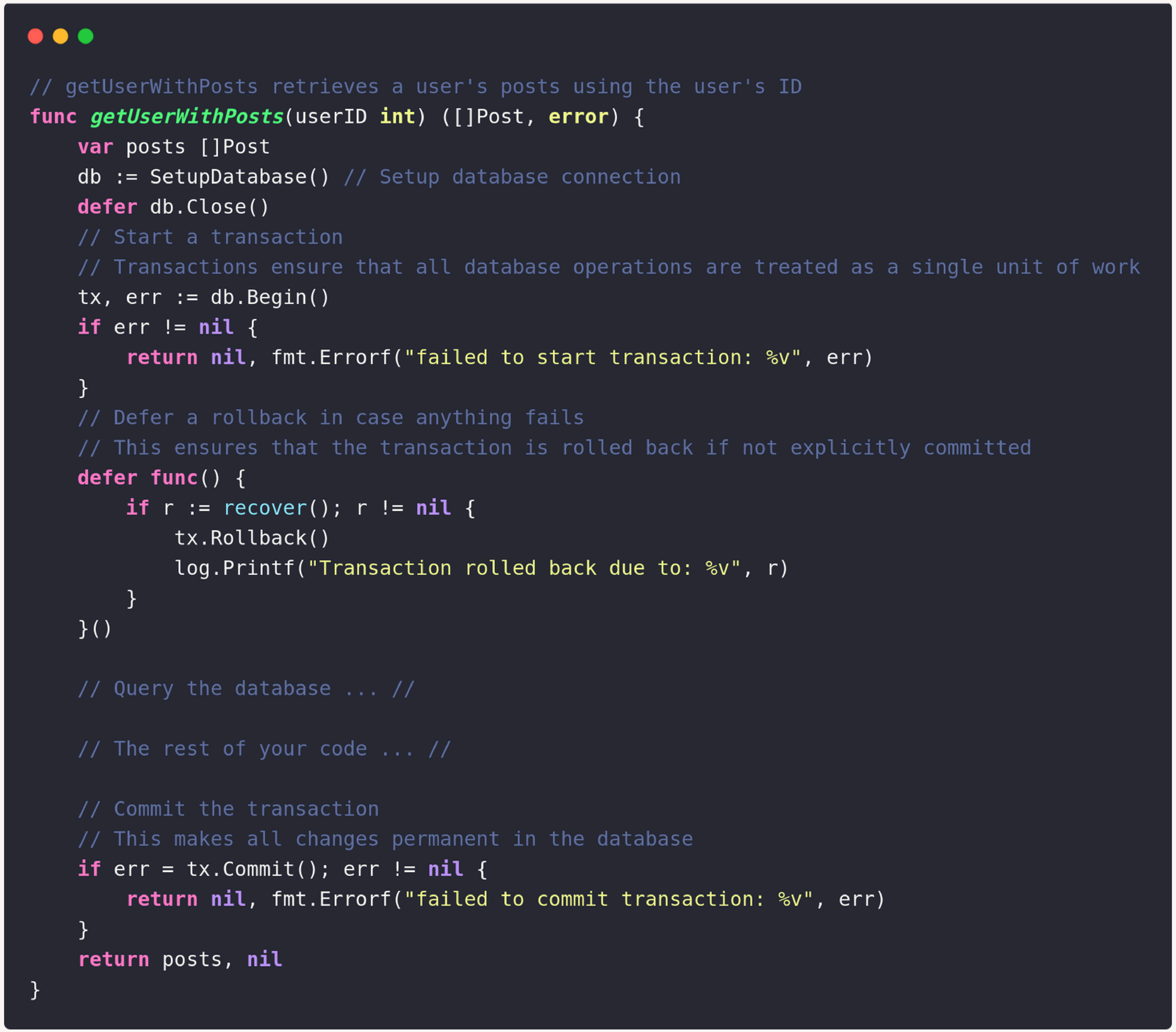
Setup Process:

1. Define SetupDatabase() function to initialize the database.
2. Create table creation functions for each entity.
3. **Implement a utility function for executing SQL statements.**

Execute SQL Statements

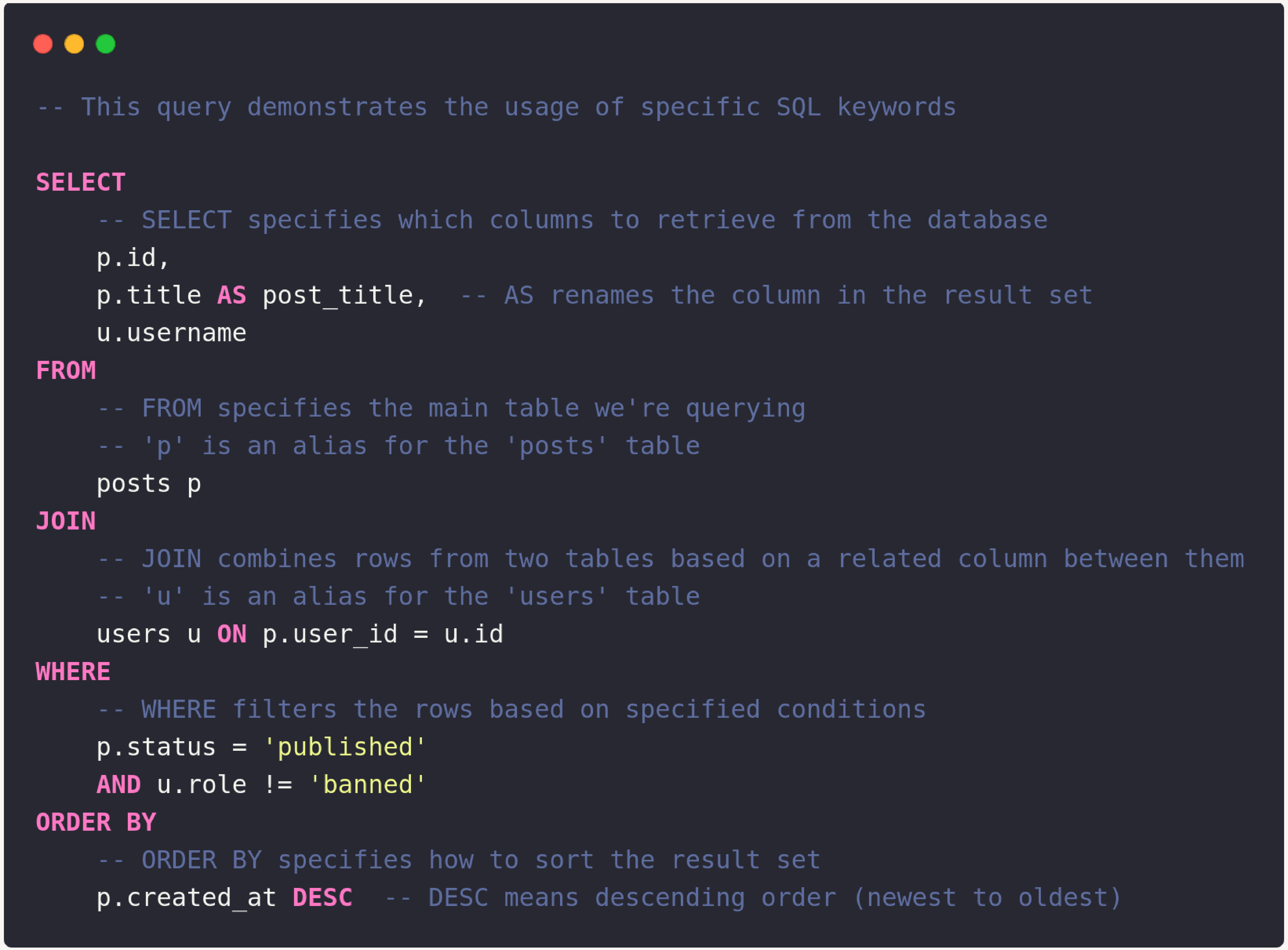
```go
// sql_exec.go
func executeSQL(db *sql.DB, sql string) {
    // Prepare the SQL statement for execution
    statement, err := db.Prepare(sql)
    if err != nil {
        log.Fatal(err) // Log and terminate on preparation error
    }

    // Execute the prepared statement
    _, err = statement.Exec()
    if err != nil {
        log.Fatal(err) // Log and terminate on execution error
    }
}
```

CRUD Function Example



```go
// getUserWithPosts retrieves a user's posts using the user's ID
func getUserWithPosts(userID int) ([]Post, error) {
    var posts []Post
    db := SetupDatabase() // Setup database connection
    defer db.Close()
    // Start a transaction
    // Transactions ensure that all database operations are treated as a single unit of work
    tx, err := db.Begin()
    if err != nil {
        return nil, fmt.Errorf("failed to start transaction: %v", err)
    }
    // Defer a rollback in case anything fails
    // This ensures that the transaction is rolled back if not explicitly committed
    defer func() {
        if r := recover(); r != nil {
            tx.Rollback()
            log.Printf("Transaction rolled back due to: %v", r)
        }
    }()

    // Query the database ... //

    // The rest of your code ... //

    // Commit the transaction
    // This makes all changes permanent in the database
    if err = tx.Commit(); err != nil {
        return nil, fmt.Errorf("failed to commit transaction: %v", err)
    }
    return posts, nil
}
```

SQL Query Keywords

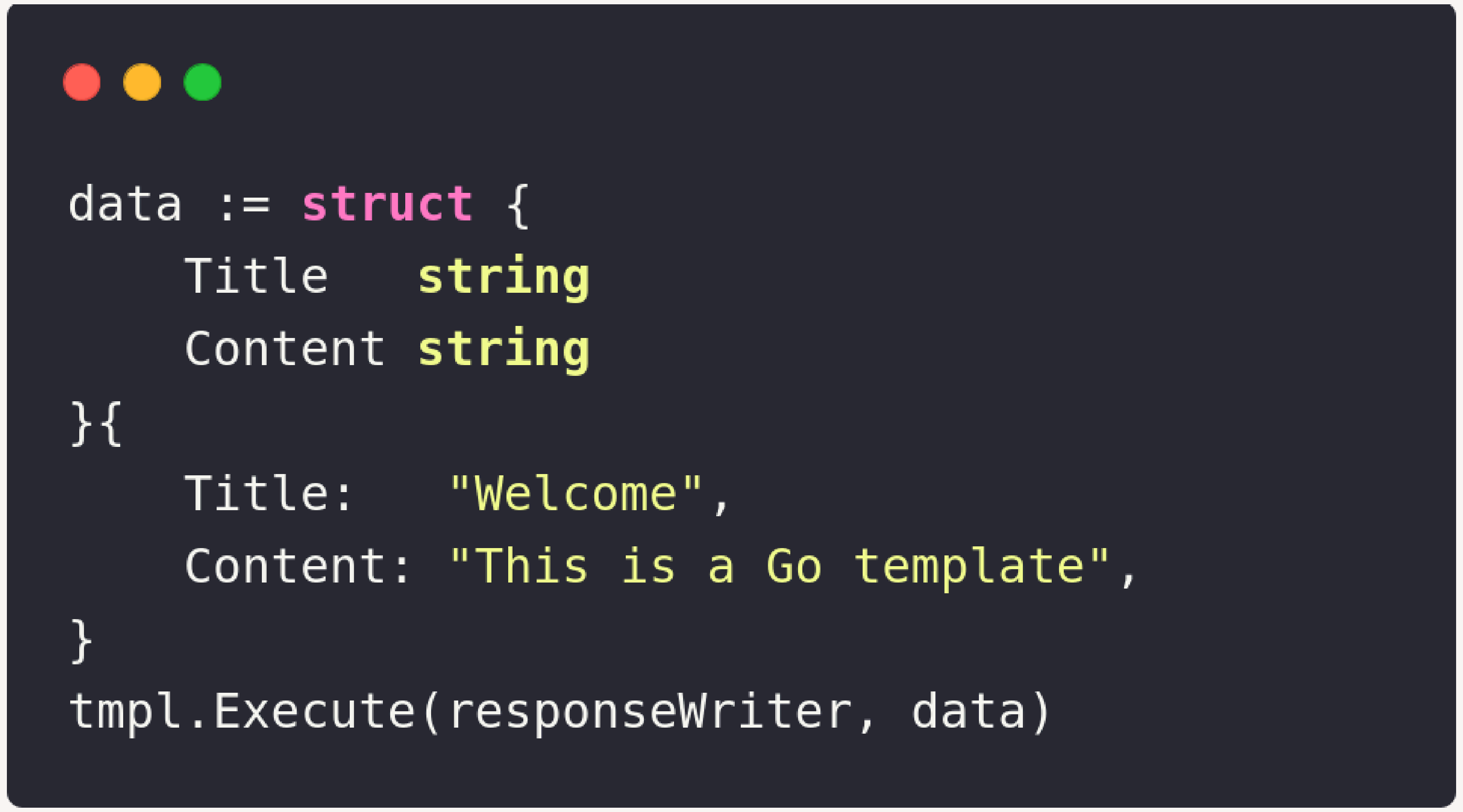


```sql
-- This query demonstrates the usage of specific SQL keywords

SELECT
    -- SELECT specifies which columns to retrieve from the database
    p.id,
    p.title AS post_title,  -- AS renames the column in the result set
    u.username
FROM
    -- FROM specifies the main table we're querying
    -- 'p' is an alias for the 'posts' table
    posts p
JOIN
    -- JOIN combines rows from two tables based on a related column between them
    -- 'u' is an alias for the 'users' table
    users u ON p.user_id = u.id
WHERE
    -- WHERE filters the rows based on specified conditions
    p.status = 'published'
    AND u.role != 'banned'
ORDER BY
    -- ORDER BY specifies how to sort the result set
    p.created_at DESC  -- DESC means descending order (newest to oldest)
```

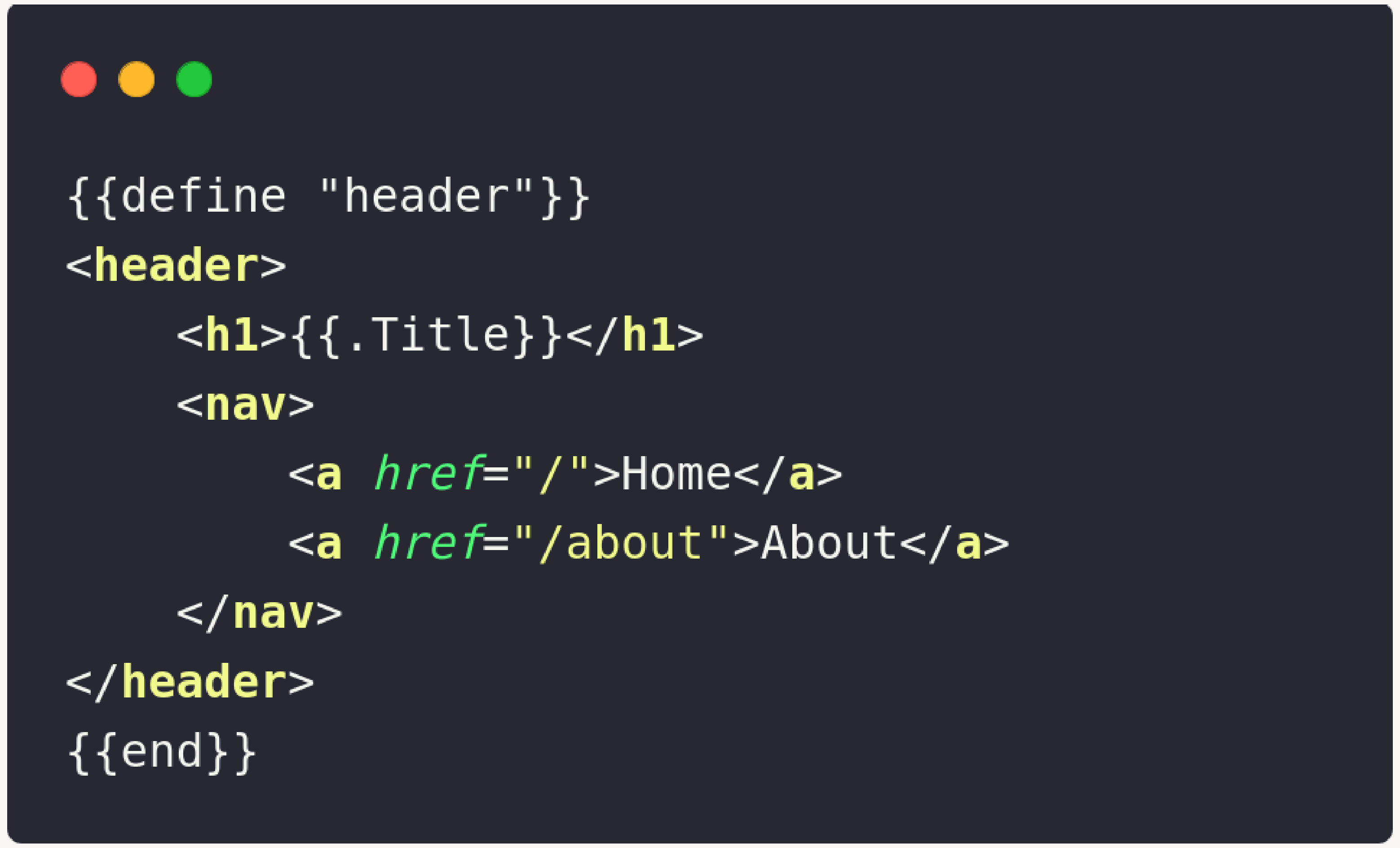
Summary

Defining HTML Templates in Go

- Templates are defined using the html/template package
- Basic syntax: {{.}} to insert a value
- You need to work with data structure inject infos.



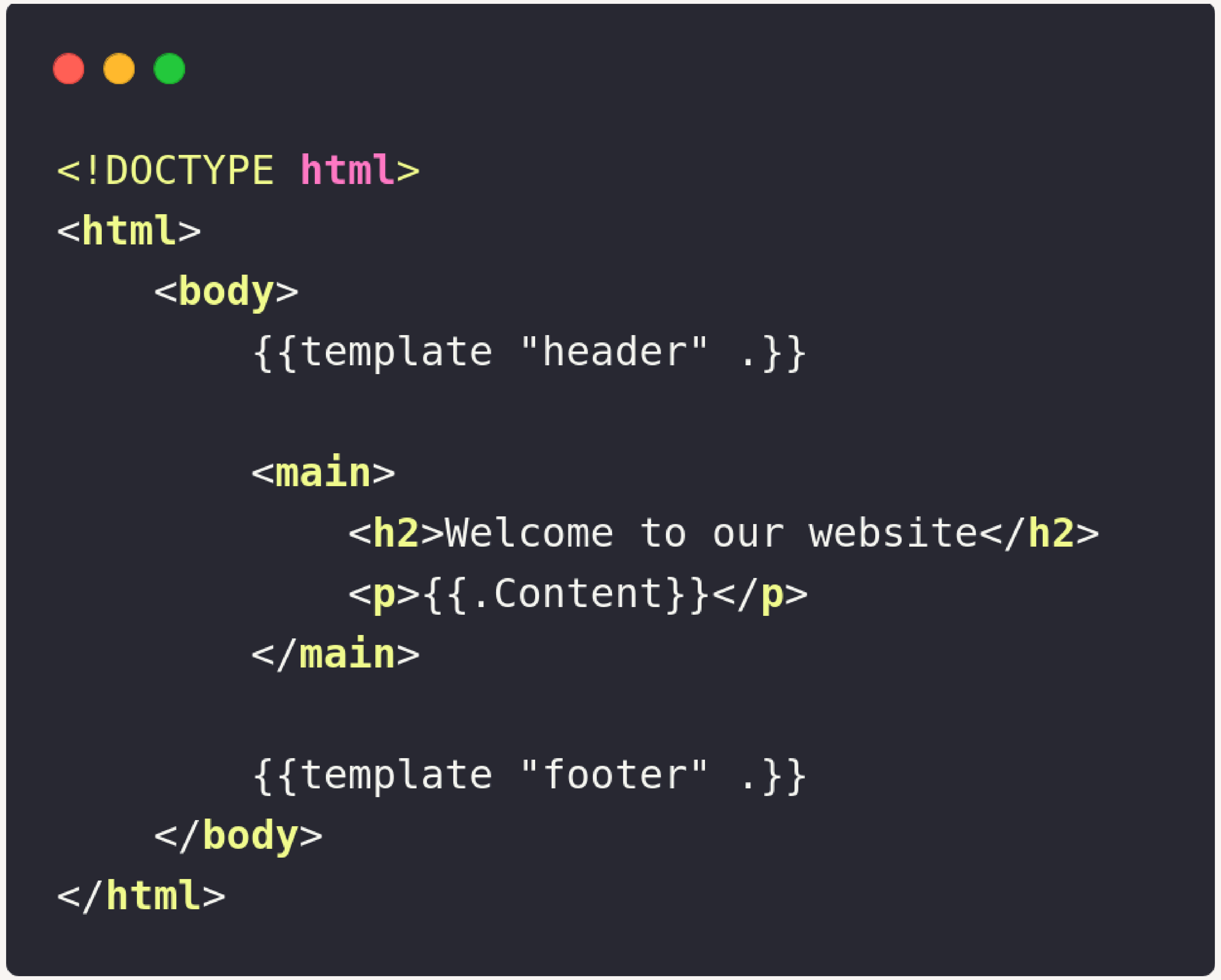
```go
data := struct {
    Title   string
    Content string
}{
    Title:   "Welcome",
    Content: "This is a Go template",
}
tmpl.Execute(responseWriter, data)
```



```html
{{define "header"}}
<header>
    <h1>{{.Title}}</h1>
    <nav>
        <a href="/">Home</a>
        <a href="/about">About</a>
    </nav>
</header>
{{end}}
```

Importing and Using Templates

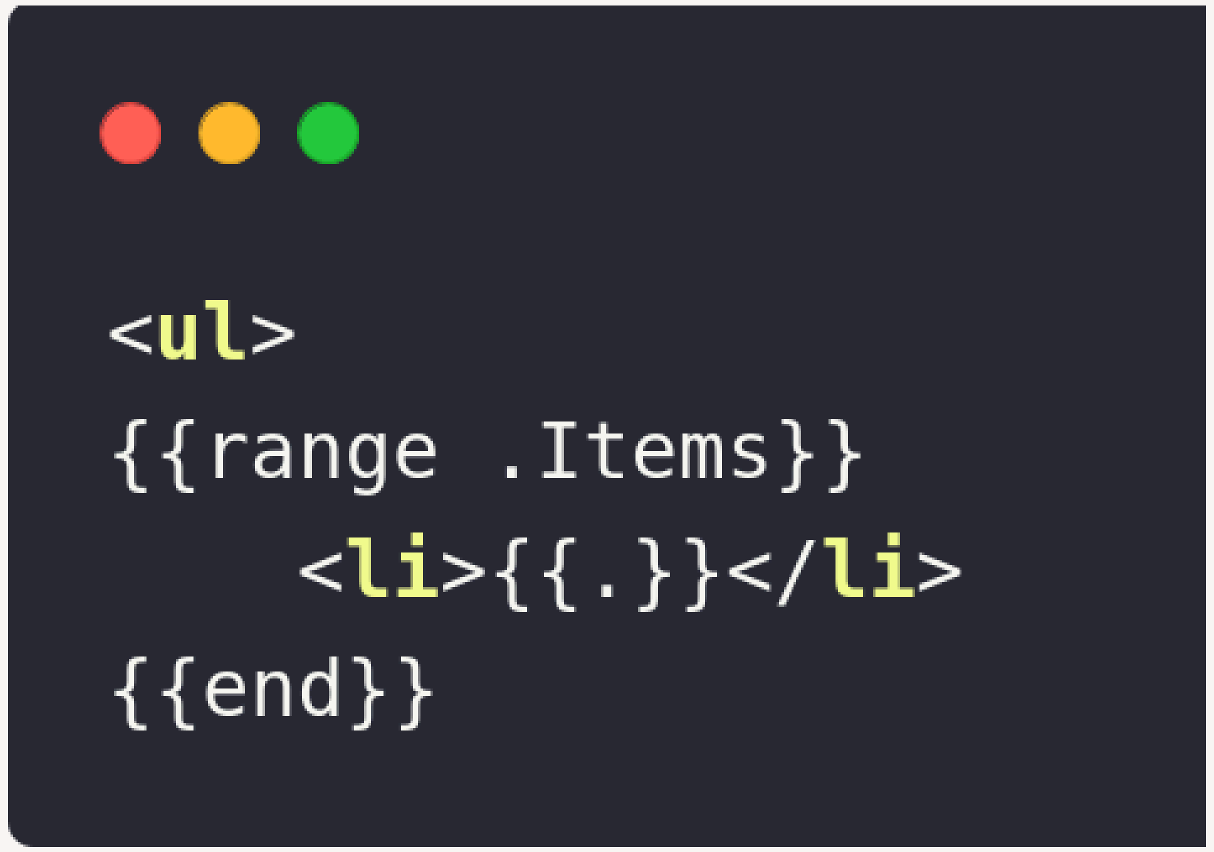
- Use {{template}} to import and use defined templates



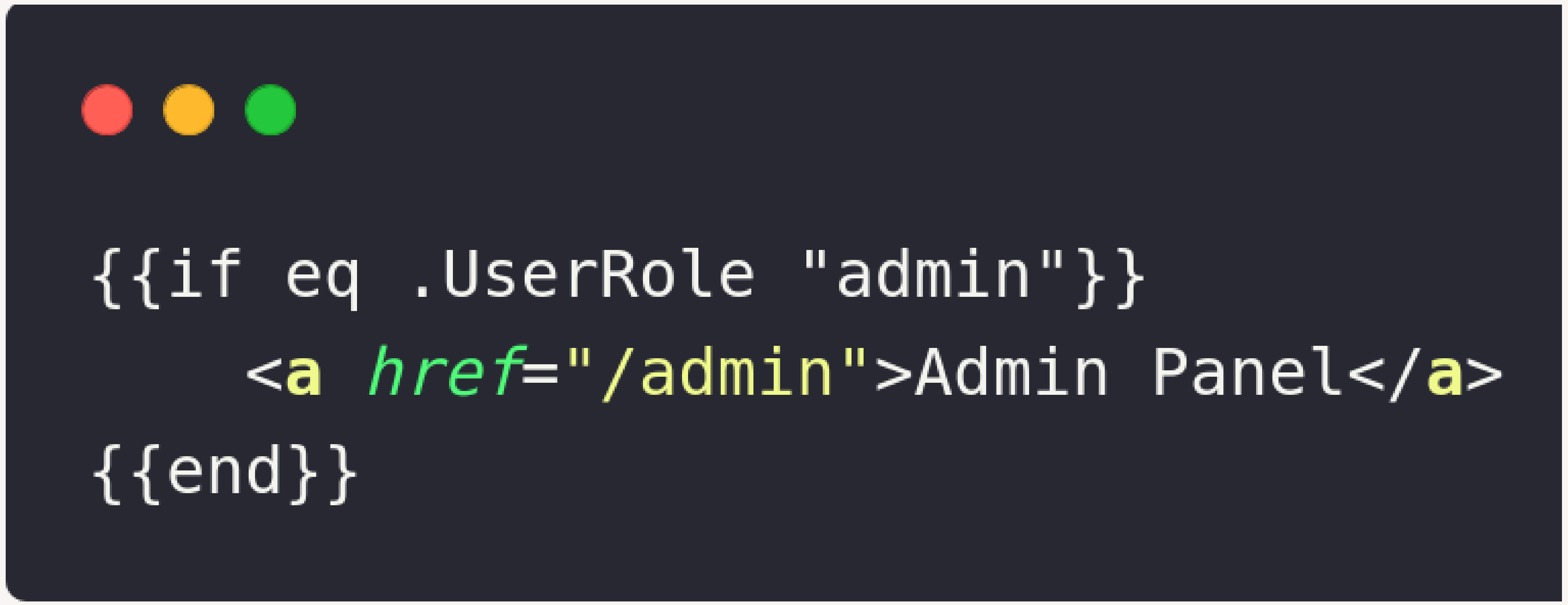
```
<!DOCTYPE html>
<html>
    <body>
        {{template "header" .}}

        <main>
            <h2>Welcome to our website</h2>
            <p>{{.Content}}</p>
        </main>

        {{template "footer" .}}
    </body>
</html>
```

Advanced Template Features

- Use range for iteration
- Use if and eq for conditionals



```
<ul>
{{range .Items}}
    <li>{{.}}</li>
{{end}}
```



```
{{if eq .UserRole "admin"}}
    <a href="/admin">Admin Panel</a>
{{end}}
```

Summary

THANKS

“Programs must be written for people to read, and only incidentally for machines to execute.”

— Harold Abelson