**1. Wprowadzenie**

**Cel projektu**

Aplikacja ma na celu zarządzanie magazynem i zasobami ludzkimi w sklepie rowerowym. Umożliwia obsługę zamówień, wniosków urlopowych, zarządzanie zadaniami oraz raportowanie. Projekt zapewnia wydajność, bezpieczeństwo i intuicyjny interfejs użytkownika.

**Zakres aplikacji**

System wspiera pracowników na różnych poziomach (administrator, dział HR, magazynier) w zarządzaniu ich obowiązkami. Oferuje intuicyjny interfejs użytkownika oraz zaawansowane funkcjonalności, takie jak generowanie raportów PDF czy szczegółowe śledzenie czasu pracy. Zastosowane rozwiązania technologiczne obejmują framework Gin, bazę danych PostgreSQL oraz konteneryzację za pomocą Dockera.

**2. Architektura systemu**

**Opis technologii**

* **Backend**: Go (Gin Framework), oferuje szybkie przetwarzanie żądań HTTP i wsparcie dla middleware.
* **Baza danych**: PostgreSQL (z wykorzystaniem ORM GORM), zapewnia relacyjną strukturę danych oraz możliwość skalowania.
* **Frontend**: HTML/CSS, z użyciem statycznych szablonów oraz responsywnego projektowania.
* **Środowisko**: Docker do zarządzania usługami aplikacji oraz łatwego wdrażania w różnych środowiskach.

**Struktura aplikacji**

* **Backend**: Obsługa logiki biznesowej oraz komunikacji z bazą danych.
* **Frontend**: Prezentacja danych oraz interakcja z użytkownikiem.
* **Middleware**: Zarządzanie sesjami i kontrola dostępu na podstawie ról użytkowników (Admin, HR, Magazyn).

**3. Moduły aplikacji**

**admin.go**

Funkcjonalność modułu:

* Zarządzanie użytkownikami, produktami i zamówieniami.
* Generowanie raportów PDF z podsumowaniem systemu.
* Przykładowe funkcje: ListUsers, CreateUser, EditUser, GenerateAdminReport.
* Uwagi: Moduł ściśle powiązany z widokami HTML, takimi jak list\_users.html i create\_user.html.

package controllers

import (

    "net/http"

    "time"

    "warehouse/models"

    "github.com/gin-gonic/gin"

    "gorm.io/gorm"

)

func AdminDashboard(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "admin\_dashboard.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    usersCount, err := models.CountUsersByPosition(db, nil, nil)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Coś poszło nie tak podczas zliczania użytkowników.",

        })

        return

    }

    incompleteTasks, err := models.CountIncompleteTasks(db, nil, nil)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia nieukończonych zadań.",

        })

        return

    }

    orders, err := models.CountOrders(db, nil, nil)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia zamówień.",

        })

        return

    }

    unpaidInvoices, err := models.CountUnpaidInvoices(db, nil, nil)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia niezapłaconych faktur.",

        })

        return

    }

    usersOnVacation, err := models.CountUsersOnVacation(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia niezapłaconych faktur.",

        })

        return

    }

    c.HTML(http.StatusOK, "admin\_dashboard.html", gin.H{

        "user\_name":                user.Name,

        "pracownik\_count":          usersCount["Magazynowy"],

        "hr\_count":                 usersCount["HR"],

        "incomplete\_tasks":         incompleteTasks,

        "orders\_count":             orders,

        "users\_vacation":           usersOnVacation,

        "unpaid\_purchase\_invoices": unpaidInvoices["UnpaidPurchaseInvoices"],

        "unpaid\_sales\_invoices":    unpaidInvoices["UnpaidSalesInvoices"],

    })

}

func AdminDashboardPost(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "admin\_dashboard.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    date1 := c.DefaultPostForm("date\_from", "")

    date2 := c.DefaultPostForm("date\_to", "")

    if date1 == "" && date2 == "" {

        AdminDashboard(c, db)

        return

    }

    dateFrom, \_ := time.Parse("2006-01-02", date1)

    dateTo, \_ := time.Parse("2006-01-02", date2)

    usersCount, err := models.CountUsersByPosition(db, &dateFrom, &dateTo)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Coś poszło nie tak podczas zliczania użytkowników.",

        })

        return

    }

    incompleteTasks, err := models.CountIncompleteTasks(db, &dateFrom, &dateTo)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia nieukończonych zadań.",

        })

        return

    }

    orders, err := models.CountOrders(db, &dateFrom, &dateTo)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia zamówień.",

        })

        return

    }

    unpaidInvoices, err := models.CountUnpaidInvoices(db, &dateFrom, &dateTo)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia niezapłaconych faktur.",

        })

        return

    }

    usersOnVacation, err := models.CountUsersOnVacation(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "admin\_dashboard.html", gin.H{

            "message": "Błąd podczas liczenia niezapłaconych faktur.",

        })

        return

    }

    c.HTML(http.StatusOK, "admin\_dashboard.html", gin.H{

        "user\_name":                user.Name,

        "pracownik\_count":          usersCount["Magazynowy"],

        "hr\_count":                 usersCount["HR"],

        "incomplete\_tasks":         incompleteTasks,

        "orders\_count":             orders,

        "users\_vacation":           usersOnVacation,

        "unpaid\_purchase\_invoices": unpaidInvoices["UnpaidPurchaseInvoices"],

        "unpaid\_sales\_invoices":    unpaidInvoices["UnpaidSalesInvoices"],

    })

    userID, exists = c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    user, err = models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "admin\_dashboard.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    c.HTML(http.StatusOK, "admin\_dashboard.html", gin.H{

        "user\_name": user.Name,

    })

}

**auth.go**

Funkcjonalność modułu:

* Obsługa logowania, rejestracji i sesji użytkownika.
* Mechanizmy logowania i wylogowania.
* Przykładowe funkcje: Login, Register, Logout, Authenticate.
* Uwagi: Zintegrowany z bcrypt w celu bezpiecznego haszowania haseł.

package controllers

import (

    "log"

    "net/http"

    "warehouse/models"

    "github.com/gin-contrib/sessions"

    "github.com/gin-gonic/gin"

    "gorm.io/gorm"

)

func Login(c \*gin.Context, db \*gorm.DB) {

    email := c.PostForm("email")

    password := c.PostForm("password")

    user, err := models.GetUserByEmail(db, email)

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            log.Printf("User not found for email: %s", email)

        } else {

            log.Printf("Error fetching user for email %s: %v", email, err)

        }

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "error": "Email lub hasło jest nieprawidłowe",

        })

        return

    }

    if user.Password != password {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "error": "Email lub hasło jest nieprawidłowe",

        })

        return

    }

    session := sessions.Default(c)

    session.Set("user\_id", user.ID)

    session.Save()

    position, \_ := models.GetUserPositionByID(db, user.ID)

    if position == "Magazynowy" {

        c.Redirect(http.StatusFound, "/warehouse/dashboard")

    } else if position == "HR" {

        c.Redirect(http.StatusFound, "/hr/dashboard")

    } else if position == "Admin" {

        c.Redirect(http.StatusFound, "/admin/dashboard")

    } else {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "error": "Twoje stanowisko jest nieprawidłowe",

        })

        return

    }

}

**human\_resources.go**

Funkcjonalność modułu:

* Zarządzanie zasobami ludzkimi: urlopami, czasem pracy, zadaniami pracowników.
* Generowanie zaświadczeń o zatrudnieniu w formacie PDF.
* Przykładowe funkcje: ListVacations, ApproveVacation, GenerateEmploymentCertificate, TrackTime.

package controllers

import (

    "fmt"

    "log"

    "net/http"

    "strconv"

    "time"

    "warehouse/models"

    "github.com/gin-gonic/gin"

    "gorm.io/gorm"

)

func HrDashboard(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "dashboard\_hr.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    vacations, \_ := models.GetAllVacations(db)

    sales\_invoices, purchase\_invoices, \_ := models.GetPendingInvoices(db)

    var purchase\_invoices\_data []map[string]interface{}

    for \_, invoice := range purchase\_invoices {

        purchase\_invoice\_data := map[string]interface{}{

            "ID":           invoice.ID,

            "SupplierName": invoice.Supplier.Name,

            "NetPrice":     invoice.NetPrice,

            "GrossPrice":   invoice.GrossPrice,

            "StatusName":   invoice.Status.Name,

        }

        purchase\_invoices\_data = append(purchase\_invoices\_data, purchase\_invoice\_data)

    }

    var sales\_invoices\_data []map[string]interface{}

    for \_, invoice := range sales\_invoices {

        sales\_invoice\_data := map[string]interface{}{

            "ID":           invoice.ID,

            "SupplierName": invoice.Customer.Name,

            "NetPrice":     invoice.NetPrice,

            "GrossPrice":   invoice.GrossPrice,

            "StatusName":   invoice.Status.Name,

        }

        sales\_invoices\_data = append(sales\_invoices\_data, sales\_invoice\_data)

    }

    var vacationsData []map[string]interface{}

    for \_, vacation := range vacations {

        vacationData := map[string]interface{}{

            "ID":        vacation.ID,

            "UserName":  vacation.User.Name + " " + vacation.User.Surname,

            "DateFrom":  vacation.DateFrom,

            "DateTo":    vacation.DateTo,

            "DateCount": vacation.DateCount,

            "Status":    vacation.Status,

        }

        vacationsData = append(vacationsData, vacationData)

    }

    c.HTML(http.StatusOK, "dashboard\_hr.html", gin.H{

        "user\_name":                 user.Name,

        "pending\_vacations":         vacationsData,

        "pending\_purchase\_invoices": purchase\_invoices\_data,

        "pending\_sales\_invoices":    sales\_invoices\_data,

    })

}

func ListWorkers(c \*gin.Context, db \*gorm.DB) {

    users, \_ := models.GetAllWorkers(db)

    c.HTML(http.StatusOK, "list\_workers.html", gin.H{"users": users})

}

func CreateWorker(c \*gin.Context, db \*gorm.DB) {

    name := c.DefaultPostForm("name", "")

    surname := c.DefaultPostForm("surname", "")

    email := c.DefaultPostForm("email", "")

    dateOfEmploymentStr := c.DefaultPostForm("date\_of\_employment", "")

    phone := c.DefaultPostForm("phone", "")

    password := c.DefaultPostForm("password", "")

    street := c.DefaultPostForm("street", "")

    city := c.DefaultPostForm("city", "")

    state := c.DefaultPostForm("state", "")

    zip := c.DefaultPostForm("zip", "")

    country := c.DefaultPostForm("country", "")

    bankAccount := c.DefaultPostForm("bank\_account", "")

    nameBank := c.DefaultPostForm("name\_bank", "")

    dateOfEmployment, err := time.Parse("2006-01-02", dateOfEmploymentStr)

    if err != nil {

        c.Redirect(http.StatusFound, "/hr/dashboard")

        c.Abort()

        return

    }

    position, \_ := models.GetPositionByName(db, "Magazynowy")

    user, err := models.CreateUser(db, name, surname, email, position, dateOfEmployment, phone, password, street, city, state, zip, country, bankAccount, nameBank)

    if err != nil {

        log.Printf("Err %s", err)

        log.Printf("User %s", user.Name)

    }

    ListWorkers(c, db)

}

func DetailWorker(c \*gin.Context, db \*gorm.DB) {

    userIDParam := c.Param("id")

    userID, \_ := strconv.ParseUint(userIDParam, 10, 64)

    userIDUint := uint(userID)

    user, \_ := models.GetUserByID(db, userIDUint)

    usersOrders, \_ := models.GetUsersOrders(db, userIDUint)

    c.HTML(http.StatusOK, "detail\_worker.html", gin.H{"user": user, "orders": usersOrders})

}

func UpdateWorkerForm(c \*gin.Context, db \*gorm.DB) {

    userIDParam := c.Param("id")

    userID, \_ := strconv.ParseUint(userIDParam, 10, 64)

    userIDUint := uint(userID)

    user, \_ := models.GetUserByID(db, userIDUint)

    c.HTML(http.StatusOK, "edit\_user.html", gin.H{

        "User": user,

    })

}

func UpdateWorker(c \*gin.Context, db \*gorm.DB) {

    userIDParam := c.Param("id")

    userID, \_ := strconv.ParseUint(userIDParam, 10, 64)

    userIDUint := uint(userID)

    user, \_ := models.GetUserByID(db, userIDUint)

    name := c.DefaultPostForm("name", user.Name)

    surname := c.DefaultPostForm("surname", user.Surname)

    email := c.DefaultPostForm("email", user.Email)

    phone := c.DefaultPostForm("phone", user.Phone)

    street := c.DefaultPostForm("street", user.Street)

    city := c.DefaultPostForm("city", user.City)

    state := c.DefaultPostForm("state", user.State)

    zip := c.DefaultPostForm("zip", user.Zip)

    country := c.DefaultPostForm("country", user.Country)

    bankAccount := c.DefaultPostForm("bank\_account", user.BankAccount)

    nameBank := c.DefaultPostForm("name\_bank", user.NameBank)

    user.Name = name

    user.Surname = surname

    user.Email = email

    user.Phone = phone

    user.Street = street

    user.City = city

    user.State = state

    user.Zip = zip

    user.Country = country

    user.BankAccount = bankAccount

    user.NameBank = nameBank

    if err := db.Save(&user).Error; err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{"error": "Błąd przy zapisywaniu zmian"})

        return

    }

    ListWorkers(c, db)

}

func DeleteWorker(c \*gin.Context, db \*gorm.DB) {

    userIDParam := c.Param("id")

    userID, \_ := strconv.ParseUint(userIDParam, 10, 64)

    userIDUint := uint(userID)

    user, \_ := models.GetUserByID(db, userIDUint)

    if err := db.Delete(&user).Error; err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{"error": "Błąd przy zapisywaniu zmian"})

        return

    }

    ListWorkers(c, db)

}

func ListOrdersHR(c \*gin.Context, db \*gorm.DB) {

    orders, \_ := models.GetAllOrders(db)

    bicycles, \_ := models.GetBicyclesNames(db)

    bicycleParts, \_ := models.GetBicyclePartsNames(db)

    users, \_ := models.GetAllWorkers(db)

    c.HTML(http.StatusOK, "list\_orders\_hr.html", gin.H{"orders": orders,

        "Bicycles":     bicycles,

        "BicycleParts": bicycleParts,

        "Users":        users})

}

func CreateOrder(c \*gin.Context, db \*gorm.DB) {

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

    bicycleIDStr := c.PostForm("bicycle\_id")

    bicyclePartIDStr := c.PostForm("bicycle\_part\_id")

    quantityStr := c.PostForm("quantity")

    userID, err := strconv.ParseUint(userIDStr, 10, 64)

    if err != nil {

        c.HTML(http.StatusBadRequest, "error.html", gin.H{"error": "Invalid user ID"})

        return

    }

    userIDUint := uint(userID)

    quantity, err := strconv.Atoi(quantityStr)

    if err != nil || quantity < 1 {

        c.HTML(http.StatusBadRequest, "error.html", gin.H{"error": "Invalid quantity"})

        return

    }

    var bicycleIDUint, bicyclePartIDUint \*uint

    if bicycleIDStr != "" {

        bicycleID, err := strconv.ParseUint(bicycleIDStr, 10, 64)

        if err != nil {

            c.HTML(http.StatusBadRequest, "error.html", gin.H{"error": "Invalid bicycle ID"})

            return

        }

        bicycleIDValue := uint(bicycleID)

        bicycleIDUint = &bicycleIDValue

    }

    // Parse bicycle part ID if selected

    if bicyclePartIDStr != "" {

        bicyclePartID, err := strconv.ParseUint(bicyclePartIDStr, 10, 64)

        if err != nil {

            c.HTML(http.StatusBadRequest, "error.html", gin.H{"error": "Invalid bicycle part ID"})

            return

        }

        bicyclePartIDValue := uint(bicyclePartID)

        bicyclePartIDUint = &bicyclePartIDValue

    }

    // Ensure at least one product is selected

    if bicycleIDUint == nil && bicyclePartIDUint == nil {

        c.HTML(http.StatusBadRequest, "error.html", gin.H{"error": "Must select either a bicycle or a bicycle part"})

        return

    }

    // Create order

    order, err := models.CreateOrder(db, userIDUint)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "error.html", gin.H{"error": "Failed to create order"})

        return

    }

    // Create order product

    orderProducts, err := models.CreateOrderProduct(db, order.ID, bicycleIDUint, bicyclePartIDUint, quantity)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "error.html", gin.H{"error": "Failed to create order product"})

        return

    }

    fmt.Printf("Order Product: %s", orderProducts.CreatedAt)

    // List orders

    ListOrdersHR(c, db)

}

func ListAplicationsHR(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "list\_applications\_hr.html", gin.H{"message": "List Applications"})

}

func DetailApplication(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "detail\_application.html", gin.H{"message": "Detail Application"})

}

func UpdateApplication(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "update\_application.html", gin.H{"message": "Update Application"})

}

func ShowReport(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "show\_report.html", gin.H{"message": "Show Report"})

}

func CreateTaskHandler(c \*gin.Context, db \*gorm.DB) {

    if c.Request.Method == http.MethodGet {

        // Pobranie listy użytkowników

        users, err := models.GetAllWorkers(db)

        if err != nil {

            c.HTML(http.StatusInternalServerError, "error.html", gin.H{

                "message": "Błąd przy pobieraniu użytkowników",

            })

            return

        }

        // Pobranie niewykonanych zadań

        tasks, err := models.GetUncompletedTasksForAllUsers(db)

        if err != nil {

            c.HTML(http.StatusInternalServerError, "error.html", gin.H{

                "message": "Błąd przy pobieraniu zadań",

            })

            return

        }

        // Obliczenie liczby dni do daty realizacji dla każdego zadania

        now := time.Now()

        type TaskWithDays struct {

            models.Task

            DaysUntilDeadline int

        }

        var tasksWithDays []TaskWithDays

        for \_, task := range tasks {

            daysUntil := int(task.Deadline.Sub(now).Hours() / 24)

            tasksWithDays = append(tasksWithDays, TaskWithDays{

                Task:              task,

                DaysUntilDeadline: daysUntil,

            })

        }

        // Renderowanie formularza i tabeli

        c.HTML(http.StatusOK, "create\_task.html", gin.H{

            "Users": users,

            "Tasks": tasksWithDays,

        })

        return

    }

    // Obsługa metody POST - tworzenie nowego zadania

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

    description := c.PostForm("description")

    deadlineStr := c.PostForm("deadline")

    priorityStr := c.PostForm("priority")

    // Parsowanie ID użytkownika

    userID, err := strconv.ParseUint(userIDStr, 10, 64)

    if err != nil {

        c.HTML(http.StatusBadRequest, "create\_task.html", gin.H{

            "message": "Nieprawidłowe ID użytkownika",

        })

        return

    }

    userIDUint := uint(userID)

    // Pobranie użytkownika z bazy

    user, err := models.GetUserByID(db, userIDUint)

    if err != nil {

        c.HTML(http.StatusBadRequest, "create\_task.html", gin.H{

            "message": "Nie znaleziono użytkownika o podanym ID",

        })

        return

    }

    // Parsowanie priorytetu

    var priority string

    switch priorityStr {

    case "1":

        priority = "Niski"

    case "2":

        priority = "Średni"

    case "3":

        priority = "Wysoki"

    default:

        c.HTML(http.StatusBadRequest, "create\_task.html", gin.H{

            "message": "Nieprawidłowy priorytet (1 - Niski, 2 - Średni, 3 - Wysoki)",

        })

        return

    }

    // Parsowanie daty deadline

    deadline, err := time.Parse("2006-01-02", deadlineStr)

    if err != nil {

        c.HTML(http.StatusBadRequest, "create\_task.html", gin.H{

            "message": "Nieprawidłowy format daty (oczekiwany: YYYY-MM-DD)",

        })

        return

    }

    // Tworzenie zadania w bazie danych

    \_, err = models.CreateTask(db, user.ID, description, deadline, priority)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "create\_task.html", gin.H{

            "message": "Błąd podczas tworzenia zadania",

        })

        return

    }

    // Pobranie zaktualizowanej listy użytkowników i zadań

    users, err := models.GetAllWorkers(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "error.html", gin.H{

            "message": "Błąd przy pobieraniu użytkowników",

        })

        return

    }

    tasks, err := models.GetUncompletedTasksForAllUsers(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "error.html", gin.H{

            "message": "Błąd przy pobieraniu zadań",

        })

        return

    }

    // Obliczenie liczby dni do daty realizacji dla każdego zadania

    now := time.Now()

    type TaskWithDays struct {

        models.Task

        DaysUntilDeadline int

    }

    // Obliczenie liczby dni do daty realizacji dla każdego zadania

    var tasksWithDays []TaskWithDays

    for \_, task := range tasks {

        daysUntil := int(task.Deadline.Sub(now).Hours() / 24)

        tasksWithDays = append(tasksWithDays, TaskWithDays{

            Task:              task,

            DaysUntilDeadline: daysUntil,

        })

    }

    // Renderowanie formularza i tabeli z komunikatem o sukcesie

    c.HTML(http.StatusOK, "create\_task.html", gin.H{

        "message": fmt.Sprintf("Zadanie dla użytkownika %s zostało pomyślnie utworzone", user.Name),

        "Users":   users,

        "Tasks":   tasksWithDays,

    })

}

func GetVacations(c \*gin.Context, db \*gorm.DB) {

    // Sprawdzenie, czy użytkownik jest zalogowany

    users, err := models.GetAllWorkers(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "error.html", gin.H{

            "message": "Błąd przy pobieraniu użytkowników",

        })

        return

    }

    // Pobranie wszystkich wniosków urlopowych z bazy danych

    vacations, err := models.GetAllVacations(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "vacation.html", gin.H{

            "error": "Nie udało się pobrać wniosków urlopowych",

        })

        return

    }

    // Renderowanie widoku z listą wniosków urlopowych

    c.HTML(http.StatusOK, "vacation.html", gin.H{

        "Vacations": vacations,

        "Users":     users,

    })

}

func UpdateVacationStatus(c \*gin.Context, db \*gorm.DB) {

    vacationIDStr := c.PostForm("vacation\_id")

    vacationID, err := strconv.Atoi(vacationIDStr)

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{"error": "Niepoprawny identyfikator urlopu"})

        return

    }

    // Pobranie nowego statusu (zatwierdzony/odrzucony) z formularza

    status := c.PostForm("status")

    if status != "Zaakceptowany" && status != "Odrzucony" {

        c.JSON(http.StatusBadRequest, gin.H{"error": "Niepoprawny status"})

        return

    }

    var rejectionReason \*string

    if status == "Odrzucony" {

        reason := c.PostForm("rejection\_reason")

        rejectionReason = &reason

    }

    // Aktualizacja statusu wniosku urlopowego

    err = models.UpdateVacationStatus(db, uint(vacationID), status, rejectionReason)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "vacation.html", gin.H{

            "error": "Nie udało się zaktualizować statusu wniosku urlopowego",

        })

        return

    }

    // Pobranie zaktualizowanej listy wniosków urlopowych

    vacations, err := models.GetAllVacations(db)

    if err != nil {

        c.HTML(http.StatusInternalServerError, "vacation.html", gin.H{

            "error": "Nie udało się pobrać wniosków urlopowych",

        })

        return

    }

    // Renderowanie widoku z zaktualizowaną listą wniosków urlopowych

    c.HTML(http.StatusOK, "vacation.html", gin.H{

        "Vacations": vacations,

        "Message":   "Status wniosku został zaktualizowany.",

    })

}

**worker.go**

Funkcjonalność modułu:

* Obsługa działań pracownika, takich jak podgląd zadań, rejestrowanie urlopów i zarządzanie czasem pracy.
* Przykładowe funkcje: WorkerDashboard, SaveVacation, GetTasks, CompleteTask.
* Uwagi: Interakcje są kierowane do pracownika, z podkreśleniem funkcji samodzielnego zarządzania zadaniami i czasem pracy.

package controllers

import (

    "bytes"

    "fmt"

    "io"

    "log"

    "net/http"

    "strconv"

    "time"

    "warehouse/models"

    "github.com/gin-gonic/gin"

    "github.com/jung-kurt/gofpdf"

    "gorm.io/gorm"

)

func WorkerDashboard(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "dashboard\_warehouse.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    // Pobierz liczbę urlopów dla użytkownika

    vacationCount, err := models.GetVacationCountByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "dashboard\_warehouse.html", gin.H{

            "message": "Coś poszło nie tak podczas sprawdzania urlopów",

        })

        return

    }

    // Przygotowanie odpowiedniego komunikatu do szablonu

    var vacationMessage string

    if vacationCount == 1 {

        vacationMessage = "Masz 1 zmianę w zakładce urlop!"

    } else if vacationCount > 1 && vacationCount <= 4 {

        vacationMessage = fmt.Sprintf("Masz %d zmiany w zakładce urlop!", vacationCount)

    } else {

        vacationMessage = fmt.Sprintf("Masz %d zmian w zakładce urlop!", vacationCount)

    }

    // Pobierz liczbę niewykonanych zadań dla użytkownika

    taskCount, err := models.GetUncompletedTasksCountByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "dashboard\_warehouse.html", gin.H{

            "message": "Coś poszło nie tak podczas sprawdzania zadań",

        })

        return

    }

    // Pobierz liczbę przeterminowanych zadań

    overdueCount, err := models.GetOverdueTasksCountByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "dashboard\_warehouse.html", gin.H{

            "message": "Coś poszło nie tak podczas sprawdzania przeterminowanych zadań",

        })

        return

    }

    // Przygotowanie odpowiedniego komunikatu do szablonu

    var taskMessage string

    if taskCount == 1 {

        taskMessage = "Masz 1 zadanie do wykonania!"

    } else if taskCount > 1 && taskCount <= 4 {

        taskMessage = fmt.Sprintf("Masz %d zadania do wykonania!", taskCount)

    } else {

        taskMessage = fmt.Sprintf("Masz %d zadań do wykonania!", taskCount)

    }

    // Przygotowanie odpowiedniego komunikatu do szablonu

    var overdueMessage string

    if overdueCount == 1 {

        overdueMessage = "Masz 1 zadanie przeterminowane do wykonania!"

    } else if overdueCount > 1 && overdueCount <= 4 {

        overdueMessage = fmt.Sprintf("Masz %d zadania przeterminowane do wykonania!", overdueCount)

    } else {

        overdueMessage = fmt.Sprintf("Masz %d zadań przeterminowanych do wykonania!", overdueCount)

    }

    c.HTML(http.StatusOK, "dashboard\_warehouse.html", gin.H{

        "user\_name":          user.Name,

        "task\_count":         taskCount,

        "task\_message":       taskMessage,

        "overdue\_count":      overdueCount,

        "overdue\_message":    overdueMessage,

        "has\_new\_tasks":      taskCount > 0,

        "has\_overdue\_tasks":  overdueCount > 0,

        "vacation\_count":     vacationCount,

        "vacation\_message":   vacationMessage,

        "has\_vacation\_tasks": vacationCount > 0,

    })

}

func DashboardWorker(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    // Pobierz dane użytkownika

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "dashboard\_worker.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    // Pobierz urlopy użytkownika

    vacations, err := models.GetVacationsByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "dashboard\_worker.html", gin.H{

            "message": "Nie udało się pobrać danych o urlopach",

        })

        return

    }

    // Przygotowanie danych o urlopach

    vacationData := []gin.H{}

    for \_, vacation := range vacations {

        vacationData = append(vacationData, gin.H{

            "id":         vacation.ID,

            "date\_from":  vacation.DateFrom,

            "date\_to":    vacation.DateTo,

            "date\_count": vacation.DateCount,

            "status":     vacation.Status,

            "reason":     vacation.RejectionReason,

            "read":       vacation.Read,

        })

    }

    // Renderowanie szablonu z listą urlopów

    c.HTML(http.StatusOK, "dashboard\_worker.html", gin.H{

        "user\_name":        user.Name,

        "user\_surname":     user.Surname,

        "user\_email":       user.Email,

        "user\_phone":       user.Phone,

        "user\_street":      user.Street,

        "user\_city":        user.City,

        "user\_state":       user.State,

        "user\_zip":         user.Zip,

        "user\_country":     user.Country,

        "user\_bankAccount": user.BankAccount,

        "user\_nameBank":    user.NameBank,

        "user\_position":    user.Position.Name,

        "user\_location":    user.City,

        "vacations":        vacationData,

    })

}

func ChangeStatus(c \*gin.Context, db \*gorm.DB) {

    // Wydrukuj otrzymane dane dla debugowania

    body, \_ := io.ReadAll(c.Request.Body)

    fmt.Printf("Otrzymane dane: %s\n", string(body))

    // Przywróć body do kontekstu

    c.Request.Body = io.NopCloser(bytes.NewBuffer(body))

    // Struktura dla danych wejściowych

    var request struct {

        VacationID int `json:"vacation\_id" binding:"required"`

    }

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

        fmt.Printf("Błąd parsowania JSON: %v\n", err)

        c.JSON(http.StatusBadRequest, gin.H{

            "success": false,

            "error": "Niepoprawne dane wejściowe: " + err.Error(),

        })

        return

    }

    // Aktualizacja statusu

    err := models.UpdateVacationStatus(db, uint(request.VacationID), "Odczytane", nil)

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "success": false,

            "error": "Nie udało się zaktualizować statusu urlopu",

        })

        return

    }

    c.JSON(http.StatusOK, gin.H{

        "success": true,

        "message": "Status został zaktualizowany",

    })

}

func SaveVacation(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    var vacationData struct {

        DateFrom  string `json:"date\_from"`

        DateTo    string `json:"date\_to"`

        DateCount int    `json:"date\_count"`

    }

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

        c.JSON(http.StatusBadRequest, gin.H{"error": "Nieprawidłowe dane wejściowe"})

        return

    }

    // Zapisz nowy urlop do bazy danych

    vacation, err := models.CreateVacation(db, userID.(uint), vacationData.DateFrom, vacationData.DateTo, vacationData.DateCount, "Wysłany")

    if err != nil {

        log.Printf("Błąd zapisywania urlopu: %v", err)

        c.JSON(http.StatusInternalServerError, gin.H{"error": "Błąd podczas zapisywania urlopu"})

        return

    }

    c.JSON(http.StatusOK, gin.H{"status": "Urlop zapisany pomyślnie", "vacation": vacation})

}

func SaveTime(c \*gin.Context, db \*gorm.DB) {

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    var workAndBreakTime struct {

        WorkedHours int `json:"worked\_hours"`

        BreakTime   int `json:"break\_time"`

    }

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

        c.JSON(http.StatusBadRequest, gin.H{"error": "Nieprawidłowe dane wejściowe"})

        return

    }

    workedHours := time.Duration(workAndBreakTime.WorkedHours) \* time.Second

    breakTime := time.Duration(workAndBreakTime.BreakTime) \* time.Second

    err := models.DailyTimekeeping(userID.(uint), workedHours, breakTime, db)

    if err != nil {

        log.Printf("Błąd zapisywania czasu pracy: %v", err)

        c.JSON(http.StatusInternalServerError, gin.H{"error": "Błąd podczas zapisywania czasu pracy"})

        return

    }

    c.JSON(http.StatusOK, gin.H{"status": "Czas pracy zapisany pomyślnie"})

}

func TimeTracking(c \*gin.Context, db \*gorm.DB) {

    userID, exists\_id := c.Get("user\_id")

    if !exists\_id {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "error": "Musisz być zalogowany",

        })

        return

    }

    daily\_report, err := models.GetDailyReportForUser(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "time\_tracking.html", gin.H{

            "message": "Błąd połączenia z bazą danych",

        })

        return

    }

    c.HTML(http.StatusOK, "time\_tracking.html", gin.H{

        "WorkHistory": daily\_report,

    })

}

func ListProducts(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "list\_products.html", gin.H{"message": "List Products"})

}

func CreateProduct(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "create\_product.html", gin.H{"message": "Create Product"})

}

func ListOrdersWorker(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "list\_orders\_worker.html", gin.H{"message": "List Orders"})

}

func DetailOrder(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "detail\_order.html", gin.H{"message": "Detail Order"})

}

func UpdateOrder(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "update\_order.html", gin.H{"message": "Update Order"})

}

func ListAplicationsWorker(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "list\_applications\_worker.html", gin.H{"message": "List Applications"})

}

func CreateApplication(c \*gin.Context, db \*gorm.DB) {

    c.HTML(http.StatusOK, "create\_application.html", gin.H{"message": "Create Application"})

}

// GenerateCertificate generates a PDF certificate with the worker's personal information

func GenerateCertificate(c \*gin.Context, db \*gorm.DB) {

    // Get the user ID from the session

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    // Get the user's data from the database

    user, err := models.GetUserByID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusUnauthorized, "dashboard\_worker.html", gin.H{

            "message": "Coś poszło nie tak",

        })

        return

    }

    // Create a new PDF document

    pdf := gofpdf.New("P", "mm", "A4", "") // A4 paper size

    // Add a page to the PDF

    pdf.AddPage()

    // Set font for the certificate

    pdf.SetFont("Arial", "B", 16)

    // Title of the certificate

    pdf.Cell(200, 10, "Zaswiadczenie o posiadaniu pracy")

    // Line break

    pdf.Ln(20)

    // Set font for content

    pdf.SetFont("Arial", "", 12)

    // Add personal data to the certificate

    pdf.Cell(190, 10, "Potwierdzamy, ze ponizsza osoba jest zatrudniona w naszej firmie:")

    pdf.Ln(10)

    pdf.Cell(100, 10, fmt.Sprintf("Imie: %s", user.Name))

    pdf.Ln(8)

    pdf.Cell(100, 10, fmt.Sprintf("Nazwisko: %s", user.Surname))

    pdf.Ln(8)

    pdf.Cell(100, 10, fmt.Sprintf("Email: %s", user.Email))

    pdf.Ln(8)

    pdf.Cell(100, 10, fmt.Sprintf("Telefon: %s", user.Phone))

    pdf.Ln(8)

    pdf.Cell(100, 10, fmt.Sprintf("Stanowisko: %s", user.Position.Name))

    pdf.Ln(8)

    pdf.Cell(100, 10, fmt.Sprintf("Lokalizacja pracy: %s", user.City))

    pdf.Ln(8)

    pdf.Cell(190, 10, "Data zatrudnienia: "+user.DateOfEmployment.Format("2006-01-02"))

    pdf.Ln(20)

    pdf.Cell(190, 10, "Wygenerowano: "+time.Now().Format("2006-01-02"))

    pdf.Ln(10)

    // Output the PDF to the browser

    c.Header("Content-Type", "application/pdf")

    c.Header("Content-Disposition", "attachment; filename=zaswiadczenie.pdf")

    err = pdf.Output(c.Writer)

    if err != nil {

        c.JSON(http.StatusInternalServerError, gin.H{

            "error": "Błąd podczas generowania PDF",

        })

    }

}

func GetTasks(c \*gin.Context, db \*gorm.DB) {

    // Sprawdzenie czy użytkownik jest zalogowany

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    // Pobranie zadań dla użytkownika z bazy danych \*TYLKO NIEWYKONANYCH\*

    tasks, err := models.GetTasksByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "worker\_task.html", gin.H{

            "error": "Nie udało się pobrać zadań",

        })

        return

    }

    // Obliczenie liczby dni do daty realizacji dla każdego zadania

    now := time.Now()

    type TaskWithDays struct {

        models.Task

        DaysUntilDeadline int

    }

    var tasksWithDays []TaskWithDays

    for \_, task := range tasks {

        daysUntil := int(task.Deadline.Sub(now).Hours() / 24)

        tasksWithDays = append(tasksWithDays, TaskWithDays{

            Task:              task,

            DaysUntilDeadline: daysUntil,

        })

    }

    // Renderowanie szablonu HTML z zadaniami

    c.HTML(http.StatusOK, "worker\_task.html", gin.H{

        "Tasks":   tasksWithDays,

        "Message": "Witaj, oto Twoje zadania.",

    })

}

func CompleteTask(c \*gin.Context, db \*gorm.DB) {

    // Pobierz taskID z formularza

    taskIDStr := c.PostForm("task\_id")

    taskID, err := strconv.Atoi(taskIDStr)

    if err != nil {

        c.JSON(http.StatusBadRequest, gin.H{"error": "Niepoprawny identyfikator zadania"})

        return

    }

    // Sprawdzenie czy użytkownik jest zalogowany

    userID, exists := c.Get("user\_id")

    if !exists {

        c.HTML(http.StatusUnauthorized, "login.html", gin.H{

            "message": "Musisz być zalogowany",

        })

        return

    }

    // Oznaczenie zadania jako wykonanego

    err = models.MarkTaskAsCompleted(db, uint(taskID), userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "worker\_task.html", gin.H{

            "error": "Nie udało się oznaczyć zadania",

        })

        return

    }

    // Pobierz zaktualizowaną listę zadań użytkownika

    tasks, err := models.GetTasksByUserID(db, userID.(uint))

    if err != nil {

        c.HTML(http.StatusInternalServerError, "worker\_task.html", gin.H{

            "error": "Nie udało się pobrać zadań",

        })

        return

    }

    // Oblicz DaysUntilDeadline dla każdego zadania

    for i := range tasks {

        tasks[i].DaysUntilDeadline = int(tasks[i].Deadline.Sub(time.Now()).Hours() / 24)

    }

    // Przekazanie zadań do widoku

    c.HTML(http.StatusOK, "worker\_task.html", gin.H{

        "Tasks": tasks,

    })

}

// func ReadVacation(c \*gin.Context, db \*gorm.DB) {

//  // Pobierz vacationID z formularza

//  vacationIDStr := c.PostForm("vacation\_id")

//  vacationID, err := strconv.Atoi(vacationIDStr)

//  if err != nil {

//      c.JSON(http.StatusBadRequest, gin.H{"error": "Niepoprawny identyfikator urlopu"})

//      return

//  }

//  // Sprawdzenie czy użytkownik jest zalogowany

//  userID, exists := c.Get("user\_id")

//  if !exists {

//      c.HTML(http.StatusUnauthorized, "login.html", gin.H{

//          "message": "Musisz być zalogowany",

//      })

//      return

//  }

//  // Oznaczenie urlopu jako odczytanego

//  err = models.MarkVacationAsRead(db, uint(vacationID))

//  if err != nil {

//      c.JSON(http.StatusInternalServerError, gin.H{

//          "error": "Nie udało się oznaczyć urlopu jako odczytanego",

//      })

//      return

//  }

//  // Pobierz zaktualizowaną listę urlopów użytkownika

//  vacations, err := models.GetVacationsByUserID(db, userID.(uint))

//  if err != nil {

//      c.HTML(http.StatusInternalServerError, "dashboard\_worker.html", gin.H{

//          "error": "Nie udało się pobrać danych o urlopach",

//      })

//      return

//  }

//  // Przekazanie urlopów do widoku

//  c.HTML(http.StatusOK, "dashboard\_worker.html", gin.H{

//      "vacations": vacations,

//  })

// }

**4. Zarządzanie danymi**

**Połączenie z bazą danych**

* Plik connection.go zawiera funkcję DatabaseConnection, która obsługuje inicjalizację połączenia z bazą PostgreSQL.
* W przypadku błędów połączenia aplikacja loguje szczegóły błędu.

package database

import (

    "log"

    "gorm.io/driver/postgres"

    "gorm.io/gorm"

)

func DatabaseConnection() (\*gorm.DB, error) {

    dsn := "host=db user=group15 password=group15 dbname=group15 port=5432 sslmode=disable TimeZone=Europe/Warsaw"

    db, err := gorm.Open(postgres.Open(dsn), &gorm.Config{})

    if err != nil {

        log.Fatalf("Błąd połączenia z bazą danych: %v", err)

        return nil, err

    }

    return db, nil

}

**Migracje**

* Plik migrations.go pozwala na inicjalizację danych w bazie, w tym przykładowych użytkowników, produktów i zamówień.
* Funkcja LoadExampleData wczytuje dane testowe, takie jak kategorie produktów, faktury i godziny pracy.

package database

import (

    "time"

    "warehouse/models"

    "gorm.io/gorm"

)

func LoadExampleData(db \*gorm.DB) {

    twoYearsAgo := time.Now().AddDate(-2, 0, 0)

    threeYearsAgo := time.Now().AddDate(-3, 0, 0)

    fiveYearsAgo := time.Now().AddDate(-5, 0, 0)

    position\_worker, \_ := models.CreatePosition(db, "Magazynowy")

    position\_hr, \_ := models.CreatePosition(db, "HR")

    position\_admin, \_ := models.CreatePosition(db, "Admin")

    worker\_1, \_ := models.CreateUser(db, "Jan", "Kowalski", "jan.kowalski@example.com", position\_worker, twoYearsAgo, "123456789", "hashed\_password1", "Chmielna 18", "Warszawa", "Mazowieckie", "05-500", "Polska", "26116055227582461851149641", "Millenium SA")

    worker\_2, \_ := models.CreateUser(db, "Anna", "Nowak", "anna.nowak@example.com", position\_worker, twoYearsAgo, "987654321", "hashed\_password2", "Kamienista 92", "Kraków", "Małopolskie", "02-365", "Polska", "30160073920644265811478567", "BNP Paribas Bank SA")

    worker\_3, \_ := models.CreateUser(db, "Piotr", "Wiśniewski", "piotr.wisniewski@example.com", position\_worker, twoYearsAgo, "555555555", "hashed\_password3", "Patrice Lumumby 16", "Łódź", "Łódzkie", "91-404", "Polska", "63147013997088337628332906", "Euro Bank SA")

    worker\_4, \_ := models.CreateUser(db, "Kamil", "Zieliński", "kamil.zielinski@example.com", position\_worker, twoYearsAgo, "666666666", "hashed\_password4", "Koścista 54", "Poznań", "Wielkopolskie", "83-452", "Polska", "59189088352449104234838308", "Pekao Bank Hipoteczny SA")

    worker\_5, \_ := models.CreateUser(db, "Marta", "Nowakowska", "marta.nowakowska@example.com", position\_worker, twoYearsAgo, "777777777", "hashed\_password5", "Sandomierska 20", "Kielce", "Świętokrzyskie", "21-347", "Polska", "64212050770960257003628031", "Santander Consumer Bank SA")

    models.CreateUser(db, "Kasia", "Wójcik", "kasia.wojcik@example.com", position\_hr, threeYearsAgo, "444444444", "hashed\_password6", "Beskidzka 15", "Bieszczady", "Podkarpackie", "35-540", "Polska", "62188001394953749136465703", "Deutshe Bank Polska SA")

    models.CreateUser(db, "Tomasz", "Kamiński", "tomasz.kaminski@example.com", position\_hr, threeYearsAgo, "333333333", "hashed\_password7", "Pijna 8", "Katowice", "Śląskie", "25-510", "Polska", "89105040426814845849987809", "ING Bank Śląski SA")

    models.CreateUser(db, "Ewa", "Zielińska", "ewa.zielinska@example.com", position\_hr, threeYearsAgo, "222222222", "hashed\_password8", "Kolejowa 88", "Wrocław", "Dolnośląskie", "34-460", "Polska", "18114000680270073853609820", "mBank SA")

    models.CreateUser(db, "Karolina", "Wiśniewska", "karolina.wisniewska@example.com", position\_hr, threeYearsAgo, "888888888", "hashed\_password9", "Centralna 1", "Gdańsk", "Pomorskie", "00-500", "Polska", "78101090585540852529024634", "Narodowy Bank Polski")

    models.CreateUser(db, "Łukasz", "Kowal", "lukasz.kowal@example.com", position\_hr, threeYearsAgo, "999999999", "hashed\_password10", "Boczna 99", "Piaseczno", "Mazowieckie", "95-530", "Polska", "44249035852272526309101433", "Alior Bank SA")

    models.CreateUser(db, "Piotr", "Mierzwa", "piotr.mierzwa@example.com", position\_admin, fiveYearsAgo, "123321123", "hashed\_password0", "Chmielna 18", "Łódź", "Łódzkie", "91-404", "Polska", "901060814452284871740455437276", "Bank BPH SA")

    address\_1, \_ := models.CreateAddress(db, "ul. Nowa 1", "Łódź", "Łódzkie", "90-001", "Polska")

    address\_2, \_ := models.CreateAddress(db, "ul. Kwiatowa 15", "Warszawa", "Mazowieckie", "00-900", "Polska")

    address\_3, \_ := models.CreateAddress(db, "ul. Słoneczna 12", "Kraków", "Małopolskie", "31-001", "Polska")

    address\_4, \_ := models.CreateAddress(db, "ul. Zielona 8", "Gdańsk", "Pomorskie", "80-001", "Polska")

    address\_5, \_ := models.CreateAddress(db, "ul. Szkolna 10", "Poznań", "Wielkopolskie", "60-001", "Polska")

    supplier\_1, \_ := models.CreateSupplier(db, "Bike Parts Supplier 1", "111223344", "supplier1@example.com", address\_1.ID)

    supplier\_2, \_ := models.CreateSupplier(db, "Bike Parts Supplier 2", "222334455", "supplier2@example.com", address\_2.ID)

    supplier\_3, \_ := models.CreateSupplier(db, "Bike Parts Supplier 3", "333445566", "supplier3@example.com", address\_3.ID)

    supplier\_4, \_ := models.CreateSupplier(db, "Bike Parts Supplier 4", "444556677", "supplier4@example.com", address\_4.ID)

    supplier\_5, \_ := models.CreateSupplier(db, "Bike Parts Supplier 5", "555667788", "supplier5@example.com", address\_5.ID)

    producer\_1, \_ := models.CreateProducer(db, "Shimano", "Japonia", "123456789", "shimano@example.com", address\_1.ID)

    producer\_2, \_ := models.CreateProducer(db, "Specialized", "USA", "987654321", "specialized@example.com", address\_2.ID)

    producer\_3, \_ := models.CreateProducer(db, "Giant", "Tajwan", "456789123", "giant@example.com", address\_3.ID)

    producer\_4, \_ := models.CreateProducer(db, "Trek", "USA", "789123456", "trek@example.com", address\_4.ID)

    producer\_5, \_ := models.CreateProducer(db, "Canyon", "Niemcy", "321654987", "canyon@example.com", address\_5.ID)

    customer\_1, \_ := models.CreateCustomer(db, "Robert", "Lewandowski", "robert.lewandowski@example.com", address\_1.ID)

    customer\_2, \_ := models.CreateCustomer(db, "Magdalena", "Kwiatkowska", "magdalena.kwiatkowska@example.com", address\_2.ID)

    customer\_3, \_ := models.CreateCustomer(db, "Jan", "Nowak", "jan.nowak@example.com", address\_3.ID)

    customer\_4, \_ := models.CreateCustomer(db, "Anna", "Wiśniewska", "anna.wisniewska@example.com", address\_4.ID)

    customer\_5, \_ := models.CreateCustomer(db, "Tomasz", "Zieliński", "tomasz.zielinski@example.com", address\_5.ID)

    category\_1, \_ := models.CreateCategoryBicycleParts(db, "Napęd")

    category\_2, \_ := models.CreateCategoryBicycleParts(db, "Koła")

    category\_3, \_ := models.CreateCategoryBicycleParts(db, "Oświetlenie")

    category\_4, \_ := models.CreateCategoryBicycleParts(db, "Hamulce")

    category\_5, \_ := models.CreateCategoryBicycleParts(db, "Amortyzatory")

    category\_bicycle\_1, \_ := models.CreateCategoryBicycle(db, "Górski")

    category\_bicycle\_2, \_ := models.CreateCategoryBicycle(db, "Szosa")

    category\_bicycle\_3, \_ := models.CreateCategoryBicycle(db, "Miejski")

    category\_bicycle\_4, \_ := models.CreateCategoryBicycle(db, "Crossowy")

    category\_bicycle\_5, \_ := models.CreateCategoryBicycle(db, "Elektryczny")

    bicyclePart\_1, \_ := models.CreateBicyclePart(db, "Korba", category\_1.ID, producer\_1.ID, 150.50, 50, 50)

    bicyclePart\_2, \_ := models.CreateBicyclePart(db, "Koło 26", category\_2.ID, producer\_2.ID, 200.00, 30, 30)

    bicyclePart\_3, \_ := models.CreateBicyclePart(db, "Lampa Przód", category\_3.ID, producer\_3.ID, 100.00, 20, 15)

    bicyclePart\_4, \_ := models.CreateBicyclePart(db, "Klocki hamulcowe", category\_4.ID, producer\_4.ID, 20.00, 100, 80)

    bicyclePart\_5, \_ := models.CreateBicyclePart(db, "Amortyzator Przód", category\_5.ID, producer\_5.ID, 500.00, 10, 5)

    bicycle\_1, \_ := models.CreateBicycle(db, "Mountain Bike", category\_bicycle\_1.ID, producer\_1.ID, 1200.00, 15, 3)

    bicycle\_2, \_ := models.CreateBicycle(db, "Road Bike", category\_bicycle\_2.ID, producer\_2.ID, 1500.00, 10, 2)

    bicycle\_3, \_ := models.CreateBicycle(db, "City Cruiser", category\_bicycle\_3.ID, producer\_2.ID, 900.00, 25, 5)

    bicycle\_4, \_ := models.CreateBicycle(db, "Hybrid Bike", category\_bicycle\_4.ID, producer\_1.ID, 1300.00, 8, 2)

    bicycle\_5, \_ := models.CreateBicycle(db, "E-Bike", category\_bicycle\_5.ID, producer\_2.ID, 2500.00, 5, 1)

    order\_1, \_ := models.CreateOrder(db, worker\_1.ID)

    order\_2, \_ := models.CreateOrder(db, worker\_2.ID)

    order\_3, \_ := models.CreateOrder(db, worker\_3.ID)

    order\_4, \_ := models.CreateOrder(db, worker\_4.ID)

    order\_5, \_ := models.CreateOrder(db, worker\_5.ID)

    models.CreateOrderProduct(db, order\_1.ID, &bicycle\_1.ID, nil, 2)

    models.CreateOrderProduct(db, order\_1.ID, nil, &bicyclePart\_1.ID, 4)

    models.CreateOrderProduct(db, order\_2.ID, &bicycle\_2.ID, nil, 1)

    models.CreateOrderProduct(db, order\_2.ID, nil, &bicyclePart\_2.ID, 6)

    models.CreateOrderProduct(db, order\_3.ID, &bicycle\_3.ID, nil, 3)

    models.CreateOrderProduct(db, order\_3.ID, nil, &bicyclePart\_3.ID, 2)

    models.CreateOrderProduct(db, order\_4.ID, &bicycle\_4.ID, nil, 4)

    models.CreateOrderProduct(db, order\_4.ID, nil, &bicyclePart\_4.ID, 5)

    models.CreateOrderProduct(db, order\_5.ID, &bicycle\_5.ID, nil, 2)

    models.CreateOrderProduct(db, order\_5.ID, nil, &bicyclePart\_5.ID, 8)

    delivery\_1, \_ := models.CreateDelivery(db, worker\_1.ID)

    delivery\_2, \_ := models.CreateDelivery(db, worker\_2.ID)

    delivery\_3, \_ := models.CreateDelivery(db, worker\_3.ID)

    delivery\_4, \_ := models.CreateDelivery(db, worker\_4.ID)

    delivery\_5, \_ := models.CreateDelivery(db, worker\_5.ID)

    models.CreateDeliveryProduct(db, delivery\_1.ID, nil, &bicyclePart\_1.ID, 10)

    models.CreateDeliveryProduct(db, delivery\_2.ID, &bicycle\_2.ID, nil, 5)

    models.CreateDeliveryProduct(db, delivery\_3.ID, nil, &bicyclePart\_3.ID, 7)

    models.CreateDeliveryProduct(db, delivery\_4.ID, &bicycle\_4.ID, nil, 3)

    models.CreateDeliveryProduct(db, delivery\_5.ID, nil, &bicyclePart\_5.ID, 12)

    status\_waiting, \_ := models.CreateStatus(db, "Oczekująca")

    status\_done, \_ := models.CreateStatus(db, "Zrealizowane")

    models.CreatePurchaseInvoice(db, supplier\_1.ID, time.Now(), 1000.00, status\_waiting, delivery\_1.ID)

    models.CreatePurchaseInvoice(db, supplier\_2.ID, twoYearsAgo, 2000.00, status\_done, delivery\_2.ID)

    models.CreatePurchaseInvoice(db, supplier\_3.ID, twoYearsAgo, 3000.00, status\_done, delivery\_2.ID)

    models.CreatePurchaseInvoice(db, supplier\_4.ID, time.Now(), 1500.00, status\_waiting, delivery\_4.ID)

    models.CreatePurchaseInvoice(db, supplier\_5.ID, threeYearsAgo, 2500.00, status\_done, delivery\_5.ID)

    models.CreateSalesInvoice(db, customer\_1.ID, order\_1.ID, time.Now(), 1200.00, status\_done)

    models.CreateSalesInvoice(db, customer\_2.ID, order\_2.ID, threeYearsAgo, 2200.00, status\_done)

    models.CreateSalesInvoice(db, customer\_3.ID, order\_3.ID, twoYearsAgo, 1800.00, status\_done)

    models.CreateSalesInvoice(db, customer\_4.ID, order\_4.ID, time.Now(), 2500.00, status\_waiting)

    models.CreateSalesInvoice(db, customer\_5.ID, order\_5.ID, threeYearsAgo, 1300.00, status\_waiting)

    models.CreateWorkingHoursDaily(db, worker\_1.ID, "2024-11-19", "8", "1")

    models.CreateWorkingHoursMonthly(db, worker\_1.ID, "2024-11", "160")

    models.CreateVacation(db, worker\_1.ID, "2024-11-19", "2024-11-26", 7, "Wysłany")

    models.CreateVacation(db, worker\_2.ID, "2024-11-19", "2024-11-26", 7, "Zaakceptowany")

    models.CreateTask(db, worker\_1.ID, "Przenieść regał NARZĘDZIA na dział WARSZTAT", time.Now(), "Niski")

    models.CreateTask(db, worker\_2.ID, "Ustawić wszystkie Eruo palety na miejsce do wywozu", time.Time{}, "Niski")

}

**5. Modele i operacje na danych**

**bicycle\_parts.go**

* **Model**: Reprezentuje części rowerowe. Pola obejmują nazwę, opis, ID roweru, ilość w magazynie.
* **Operacje**: Dodawanie części rowerowych, aktualizacja stanu magazynowego, usuwanie części.

package models

import (

    "gorm.io/gorm"

)

type CategoryBicycleParts struct {

    gorm.Model

    Name          string          `gorm:"unique;not null"`

}

type BicyclePart struct {

    gorm.Model

    Name                 string    `gorm:"not null"`

    CategoryID           uint      `gorm:"not null"`

    Category             CategoryBicycleParts

    ProducerID           uint      `gorm:"not null"`

    Producer             Producer

    Price                float32   `gorm:"not null"`

    Quantity             int       `gorm:"not null"`

    LastQuantityAdded    int       `gorm:"not null"`

}

func CreateCategoryBicycleParts(db \*gorm.DB, name string) (CategoryBicycleParts, error) {

    category := CategoryBicycleParts{

        Name: name,

    }

    result := db.Create(&category)

    return category, result.Error

}

func CreateBicyclePart(db \*gorm.DB, name string, categoryID uint, producerID uint, price float32, quantity, lastQuantityAdded int) (BicyclePart, error) {

    bicyclePart := BicyclePart{

        Name:              name,

        CategoryID:        categoryID,

        ProducerID:        producerID,

        Price:             price,

        Quantity:          quantity,

        LastQuantityAdded: lastQuantityAdded,

    }

    result := db.Create(&bicyclePart)

    return bicyclePart, result.Error

}

func GetBicyclePartsNames(db \*gorm.DB) ([]BicyclePart, error) {

    var bicyclePartNames []BicyclePart

    if err := db.Find(&bicyclePartNames).Error; err != nil {

        return nil, err

    }

    return bicyclePartNames, nil

}

**bicycle.go**

* **Model**: Reprezentuje rowery. Pola obejmują nazwę, kategorię, cenę, wagę.
* **Operacje**: Zarządzanie rowerami – dodawanie, edycja, usuwanie, filtrowanie według kategorii.

package models

import "gorm.io/gorm"

type CategoryBicycle struct {

    gorm.Model

    Name     string    `gorm:"unique;not null"`

}

type Bicycle struct {

    gorm.Model

    Name              string `gorm:"not null"`

    CategoryID        uint   `gorm:"not null"`

    Category          CategoryBicycle

    ProducerID        uint   `gorm:"not null"`

    Producer          Producer

    Price             float32 `gorm:"not null"`

    Quantity          int     `gorm:"not null"`

    LastQuantityAdded int     `gorm:"not null"`

}

func CreateCategoryBicycle(db \*gorm.DB, name string) (CategoryBicycle, error) {

    category := CategoryBicycle{

        Name: name,

    }

    result := db.Create(&category)

    return category, result.Error

}

func CreateBicycle(db \*gorm.DB, name string, categoryID, producerID uint, price float32, quantity, lastQuantityAdded int) (Bicycle, error) {

    bicycle := Bicycle{

        Name:              name,

        CategoryID:        categoryID,

        ProducerID:        producerID,

        Price:             price,

        Quantity:          quantity,

        LastQuantityAdded: lastQuantityAdded,

    }

    result := db.Create(&bicycle)

    return bicycle, result.Error

}

func GetBicyclesNames(db \*gorm.DB) ([]Bicycle, error) {

    var bicyclesNames []Bicycle

    if err := db.Find(&bicyclesNames).Error; err != nil {

        return nil, err

    }

    return bicyclesNames, nil

}

**contractors.go**

* **Model**: Reprezentuje kontrahentów (np. dostawców, klientów). Pola obejmują nazwę, adres, NIP.
* **Operacje**: Dodawanie kontrahentów, aktualizacja danych, usuwanie.

package models

import "gorm.io/gorm"

type Address struct {

    gorm.Model

    Street     string `gorm:"not null"`

    City       string `gorm:"not null"`

    State      string

    PostalCode string `gorm:"not null"`

    Country    string `gorm:"not null"`

}

type Supplier struct {

    gorm.Model

    Name      string `gorm:"not null"`

    Phone     string `gorm:"not null"`

    Email     string `gorm:"not null"`

    AddressID uint

    Address   Address

}

type Producer struct {

    gorm.Model

    Name      string `gorm:"not null"`

    Country   string `gorm:"not null"`

    Phone     string `gorm:"not null"`

    Email     string `gorm:"not null"`

    AddressID uint

    Address   Address

}

type Customer struct {

    gorm.Model

    Name      string `gorm:"not null"`

    Phone     string `gorm:"not null"`

    Email     string `gorm:"not null"`

    AddressID uint

    Address   Address `gorm:"foreignKey:AddressID"`

}

func CreateAddress(db \*gorm.DB, street string, city string, state string, postalCode string, country string) (Address, error) {

    address := Address{

        Street:     street,

        City:       city,

        State:      state,

        PostalCode: postalCode,

        Country:    country,

    }

    result := db.Create(&address)

    return address, result.Error

}

func CreateSupplier(db \*gorm.DB, name string, phone string, email string, addressID uint) (Supplier, error) {

    supplier := Supplier{

        Name:      name,

        Phone:     phone,

        Email:     email,

        AddressID: addressID,

    }

    result := db.Create(&supplier)

    return supplier, result.Error

}

func CreateProducer(db \*gorm.DB, name string, country string, phone string, email string, addressID uint) (Producer, error) {

    producer := Producer{

        Name:      name,

        Country:   country,

        Phone:     phone,

        Email:     email,

        AddressID: addressID,

    }

    result := db.Create(&producer)

    return producer, result.Error

}

func CreateCustomer(db \*gorm.DB, name string, phone string, email string, addressID uint) (Customer, error) {

    client := Customer{

        Name:      name,

        Phone:     phone,

        Email:     email,

        AddressID: addressID,

    }

    result := db.Create(&client)

    return client, result.Error

}

**deliveries.go**

* **Model**: Obejmuje dostawy, ich datę, status oraz powiązane zamówienia.
* **Operacje**: Tworzenie dostaw, zmiana statusu, usuwanie dostaw.

package models

import "gorm.io/gorm"

type Delivery struct {

    gorm.Model

    UserID        uint

    User          User           `gorm:"foreignKey:UserID"`

}

type DeliveryProduct struct {

    gorm.Model

    OrderID        uint

    Order          Order       `gorm:"foreignKey:OrderID"`

    BicycleID      \*uint

    Bicycle        \*Bicycle    `gorm:"foreignKey:BicycleID"`

    BicyclePartID  \*uint

    BicyclePart    \*BicyclePart `gorm:"foreignKey:BicyclePartID"`

    Quantity       int

}

func CreateDelivery(db \*gorm.DB, userID uint) (Delivery, error) {

    delivery := Delivery{

        UserID:        userID,

    }

    result := db.Create(&delivery)

    return delivery, result.Error

}

func CreateDeliveryProduct(db \*gorm.DB, orderID uint, bicycleID, bicyclePartID \*uint, quantity int) (DeliveryProduct, error) {

    deliveryProduct := DeliveryProduct{

        OrderID:        orderID,

        BicycleID:      bicycleID,

        BicyclePartID:  bicyclePartID,

        Quantity:       quantity,

    }

    result := db.Create(&deliveryProduct)

    return deliveryProduct, result.Error

}

**hours.go**

* **Model**: Rejestruje godziny pracy użytkowników. Pola obejmują datę, liczbę godzin, status (zatwierdzone/oczekujące).
* **Operacje**: Rejestracja godzin pracy, aktualizacja wpisów, usuwanie.

package models

import (

    "log"

    "time"

    "gorm.io/gorm"

    "warehouse/helpers"

)

type WorkingHoursDaily struct {

    gorm.Model

    UserID      uint   `gorm:"not null"`

    User        User   `gorm:"foreignKey:UserID"`

    Day         string `gorm:"not null"`

    WorkedHours string `gorm:"type:varchar(8);not null"`

    BreakTime   string `gorm:"type:varchar(8);not null"`

}

type WorkingHoursMonthly struct {

    gorm.Model

    UserID           uint   `gorm:"not null"`

    User             User   `gorm:"foreignKey:UserID"`

    Month            string `gorm:"unique;not null"`

    TotalWorkedHours string `gorm:"type:varchar(8);not null"`

}

func CreateWorkingHoursDaily(db \*gorm.DB, userID uint, day, workedHours, breakTime string) (WorkingHoursDaily, error) {

    workingHours := WorkingHoursDaily{

        UserID:      userID,

        Day:         day,

        WorkedHours: workedHours,

        BreakTime:   breakTime,

    }

    result := db.Create(&workingHours)

    return workingHours, result.Error

}

func CreateWorkingHoursMonthly(db \*gorm.DB, userID uint, month, totalWorkedHours string) (WorkingHoursMonthly, error) {

    workingHours := WorkingHoursMonthly{

        UserID:           userID,

        Month:            month,

        TotalWorkedHours: totalWorkedHours,

    }

    result := db.Create(&workingHours)

    return workingHours, result.Error

}

func DailyTimekeeping(user\_id uint, worked\_hours time.Duration, break\_time time.Duration, db \*gorm.DB) error {

    today := time.Now().Format("2006-01-02")

    workedHoursStr := helpers.FormatDurationToHHMMSS(worked\_hours)

    breakTimeStr := helpers.FormatDurationToHHMMSS(break\_time)

    daily\_time := WorkingHoursDaily{

        UserID:      user\_id,

        Day:         today,

        WorkedHours: workedHoursStr,

        BreakTime:   breakTimeStr,

    }

    log.Printf("Zapisujemy czas pracy: UserID: %d, Dzień: %s, WorkedHours: %s, BreakTime: %s", user\_id, today, workedHoursStr, breakTimeStr)

    result := db.Create(&daily\_time)

    if result.Error != nil {

        log.Printf("Błąd podczas zapisywania rekordu czasu pracy: %v", result.Error)

        return result.Error

    }

    log.Println("Rekord czasu pracy zapisany pomyślnie")

    return nil

}

func MonthlyTimekeeping(user\_id uint, time\_duration string, db \*gorm.DB) error {

    var monthlyHours WorkingHoursMonthly

    month := time.Now().Format("2006-01")

    result := db.Where("user\_id = ? AND month = ?", user\_id, month).First(&monthlyHours)

    if result.Error == gorm.ErrRecordNotFound {

        monthly\_timekeeping := WorkingHoursMonthly{

            UserID:           user\_id,

            Month:            month,

            TotalWorkedHours: time\_duration,

        }

        db.Create(&monthly\_timekeeping)

    } else {

        monthlyHours.TotalWorkedHours += time\_duration

        result = db.Save(&monthlyHours)

    }

    return result.Error

}

func GetDailyReportForUser(db \*gorm.DB, userID uint) ([]WorkingHoursDaily, error) {

    var workingHours []WorkingHoursDaily

    result := db.Where("user\_id = ?", userID).Find(&workingHours)

    if result.Error != nil {

        log.Printf("Błąd podczas pobierania danych godzin pracy dla użytkownika %d: %v", userID, result.Error)

        return nil, result.Error

    }

    return workingHours, nil

}

**invoices.go**

* **Model**: Reprezentuje faktury z polami takimi jak numer faktury, kwota, status.
* **Operacje**: Tworzenie faktur, zmiana statusu (opłacone/nieopłacone), usuwanie.

package models

import (

    "time"

    "gorm.io/gorm"

)

type Status struct {

    gorm.Model

    Name string `gorm:"not null"`

}

type PurchaseInvoice struct {

    gorm.Model

    SupplierID uint      `gorm:"not null"`

    Supplier   Supplier  `gorm:"foreignKey:SupplierID"`

    IssueDate  time.Time `gorm:"not null"`

    NetPrice   float32   `gorm:"not null"`

    GrossPrice float32   `gorm:"not null"`

    StatusID   uint      `gorm:"foreignKey:StatusID"`

    Status     Status    `gorm:"not null"`

    DeliveryID uint      `gorm:"not null"`

    Delivery   Delivery  `gorm:"foreignKey:DeliveryID"`

}

type SalesInvoice struct {

    gorm.Model

    CustomerID uint      `gorm:"not null"`

    Customer   Customer  `gorm:"foreignKey:CustomerID"`

    OrderID    uint      `gorm:"not null"`

    Order      Order     `gorm:"foreignKey:OrderID"`

    IssueDate  time.Time `gorm:"not null"`

    NetPrice   float32   `gorm:"not null"`

    GrossPrice float32   `gorm:"not null"`

    StatusID   uint      `gorm:"foreignKey:StatusID"`

    Status     Status    `gorm:"not null"`

}

func CreateStatus(db \*gorm.DB, name string) (Status, error) {

    status := Status{

        Name: name,

    }

    result := db.Create(&status)

    return status, result.Error

}

func CreatePurchaseInvoice(db \*gorm.DB, supplierID uint, issueDate time.Time, netPrice float32, status Status, deliveryID uint) (PurchaseInvoice, error) {

    purchaseInvoice := PurchaseInvoice{

        SupplierID: supplierID,

        IssueDate:  issueDate,

        NetPrice:   netPrice,

        GrossPrice: netPrice \* 1.23,

        Status:     status,

        DeliveryID: deliveryID,

    }

    result := db.Create(&purchaseInvoice)

    return purchaseInvoice, result.Error

}

func CreateSalesInvoice(db \*gorm.DB, customerID, orderID uint, issueDate time.Time, netPrice float32, status Status) (SalesInvoice, error) {

    salesInvoice := SalesInvoice{

        CustomerID: customerID,

        OrderID:    orderID,

        IssueDate:  issueDate,

        NetPrice:   netPrice,

        GrossPrice: netPrice \* 1.23,

        Status:     status,

    }

    result := db.Create(&salesInvoice)

    return salesInvoice, result.Error

}

func GetPendingInvoices(db \*gorm.DB) ([]SalesInvoice, []PurchaseInvoice, error) {

    var sales\_invoices []SalesInvoice

    var purchase\_invoices []PurchaseInvoice

    err\_sales := db.Preload("Supplier").Preload("Status").Where("status\_id = ?", 1).Find(&purchase\_invoices).Error

    err\_purchase := db.Preload("Customer").Preload("Status").Where("status\_id = ?", 1).Find(&sales\_invoices).Error

    if err\_sales != nil {

        if err\_purchase != nil {

            return nil, nil, err\_sales

        }

        return nil, purchase\_invoices, err\_sales

    }

    return sales\_invoices, purchase\_invoices, nil

}

func CountUnpaidInvoices(db \*gorm.DB, dateFrom, dateTo \*time.Time) (map[string]int64, error) {

    metrics := make(map[string]int64)

    // Zmienna tymczasowa dla faktur zakupowych

    var unpaidPurchase int64

    query := db.Model(&PurchaseInvoice{}).Where("status\_id = ?", 1)

    if dateFrom != nil && dateTo != nil {

        query = query.Where("issue\_date BETWEEN ? AND ?", \*dateFrom, \*dateTo)

    } else if dateFrom != nil {

        query = query.Where("issue\_date >= ?", \*dateFrom)

    } else if dateTo != nil {

        query = query.Where("issue\_date <= ?", \*dateTo)

    }

    if err := query.Count(&unpaidPurchase).Error; err != nil {

        return nil, err

    }

    metrics["UnpaidPurchaseInvoices"] = unpaidPurchase

    // Zmienna tymczasowa dla faktur sprzedażowych

    var unpaidSales int64

    query = db.Model(&SalesInvoice{}).Where("status\_id = ?", 1)

    if dateFrom != nil && dateTo != nil {

        query = query.Where("issue\_date BETWEEN ? AND ?", \*dateFrom, \*dateTo)

    } else if dateFrom != nil {

        query = query.Where("issue\_date >= ?", \*dateFrom)

    } else if dateTo != nil {

        query = query.Where("issue\_date <= ?", \*dateTo)

    }

    if err := query.Count(&unpaidSales).Error; err != nil {

        return nil, err

    }

    metrics["UnpaidSalesInvoices"] = unpaidSales

    return metrics, nil

}

**orders.go**

* **Model**: Rejestruje zamówienia. Pola obejmują ID użytkownika, roweru, ilość, datę zamówienia.
* **Operacje**: Zarządzanie zamówieniami – tworzenie, edycja, zmiana statusu, usuwanie.

package models

import (

    "time"

    "gorm.io/gorm"

)

type Order struct {

    gorm.Model

    UserID uint

    User   User `gorm:"foreignKey:UserID"`

}

type OrderProduct struct {

    gorm.Model

    OrderID       uint

    Order         Order `gorm:"foreignKey:OrderID"`

    BicycleID     \*uint

    Bicycle       \*Bicycle `gorm:"foreignKey:BicycleID"`

    BicyclePartID \*uint

    BicyclePart   \*BicyclePart `gorm:"foreignKey:BicyclePartID"`

    Quantity      int

}

func CreateOrder(db \*gorm.DB, userID uint) (Order, error) {

    order := Order{

        UserID: userID,

    }

    result := db.Create(&order)

    return order, result.Error

}

func CreateOrderProduct(db \*gorm.DB, orderID uint, bicycleID, bicyclePartID \*uint, quantity int) (OrderProduct, error) {

    orderProduct := OrderProduct{

        OrderID:       orderID,

        BicycleID:     bicycleID,

        BicyclePartID: bicyclePartID,

        Quantity:      quantity,

    }

    result := db.Create(&orderProduct)

    return orderProduct, result.Error

}

func GetAllOrders(db \*gorm.DB) ([]OrderProduct, error) {

    var orders []OrderProduct

    if err := db.Preload("Bicycle").Preload("BicyclePart").Find(&orders).Error; err != nil {

        return nil, err

    }

    return orders, nil

}

func GetUsersOrders(db \*gorm.DB, userID uint) ([]OrderProduct, error) {

    var orders []OrderProduct

    err := db.Where("orders.user\_id = ?", userID).

        Joins("JOIN orders ON orders.id = order\_products.order\_id").

        Preload("Bicycle").

        Preload("BicyclePart").

        Find(&orders).Error

    if err != nil {

        return nil, err

    }

    return orders, nil

}

func CountOrders(db \*gorm.DB, dateFrom, dateTo \*time.Time) (int64, error) {

    var orderCount int64

    // Zbuduj zapytanie z opcjonalnym filtrowaniem dat

    query := db.Model(&Order{})

    if dateFrom != nil && dateTo != nil {

        query = query.Where("created\_at BETWEEN ? AND ?", \*dateFrom, \*dateTo)

    } else if dateFrom != nil {

        query = query.Where("created\_at >= ?", \*dateFrom)

    } else if dateTo != nil {

        query = query.Where("created\_at <= ?", \*dateTo)

    }

    // Zlicz zamówienia

    if err := query.Count(&orderCount).Error; err != nil {

        return 0, err

    }

    return orderCount, nil

}

**task.go**

* **Model**: Przechowuje informacje o zadaniach pracowników, w tym opis, priorytet, status.
* **Operacje**: Dodawanie zadań, oznaczanie jako wykonane, filtrowanie zadań opóźnionych.

package models

import (

    "time"

    "gorm.io/gorm"

)

type Task struct {

    gorm.Model

    UserID            uint      `gorm:"not null"`          // ID pracownika, któremu przypisano zadanie

    User              User      `gorm:"foreignKey:UserID"` // Relacja z modelem User

    Description       string    `gorm:"not null"`          // Opis zadania

    Deadline          time.Time `gorm:"not null"`          // Data realizacji zadania

    Priority          string    `gorm:"not null"`          // Priorytet zadania (np. 1 = wysoki, 2 = średni, 3 = niski)

    IsCompleted       bool      `gorm:"default:false"`     // Status zadania (false = niewykonane, true = wykonane)

    DaysUntilDeadline int       `gorm:"-"`                 // Pole obliczane dynamicznie

}

// Tworzenie nowego zadania

func CreateTask(db \*gorm.DB, userID uint, description string, deadline time.Time, priority string) (Task, error) {

    task := Task{

        UserID:      userID,

        Description: description,

        Deadline:    deadline,

        Priority:    priority,

        IsCompleted: false,

    }

    result := db.Create(&task)

    return task, result.Error

}

// Pobieranie niewykonanych zadań przypisanych do pracownika

func GetTasksByUserID(db \*gorm.DB, userID uint) ([]Task, error) {

    var tasks []Task

    err := db.Where("user\_id = ?", userID).Where("is\_completed = ?", false).Order("deadline ASC, priority ASC").Find(&tasks).Error

    return tasks, err

}

// Oznaczanie zadania jako wykonane

func MarkTaskAsCompleted(db \*gorm.DB, taskID uint, UserID uint) error {

    return db.Model(&Task{}).Where("id = ?", taskID).Update("is\_completed", true).Error

}

// Pobieranie niewykonanych zadań pracowników

func GetUncompletedTasksForAllUsers(db \*gorm.DB) ([]Task, error) {

    var tasks []Task

    err := db.Preload("User").Where("is\_completed = ?", false).Find(&tasks).Error

    return tasks, err

}

// Pobieranie liczby niewykonanych zadań dla użytkownika

func GetUncompletedTasksCountByUserID(db \*gorm.DB, userID uint) (int64, error) {

    var count int64

    err := db.Model(&Task{}).Where("user\_id = ? AND is\_completed = ?", userID, false).Count(&count).Error

    return count, err

}

func GetOverdueTasksCountByUserID(db \*gorm.DB, userID uint) (int, error) {

    var count int64

    err := db.Model(&Task{}).

        Where("user\_id = ? AND is\_completed = ? AND deadline < ?", userID, false, time.Now()).

        Count(&count).Error

    return int(count), err

}

func CountIncompleteTasks(db \*gorm.DB, dateFrom, dateTo \*time.Time) (int64, error) {

    var incompleteTaskCount int64

    // Zbuduj zapytanie z opcjonalnym filtrowaniem dat

    query := db.Model(&Task{}).Where("is\_completed = ?", false)

    if dateFrom != nil && dateTo != nil {

        query = query.Where("deadline BETWEEN ? AND ?", \*dateFrom, \*dateTo)

    } else if dateFrom != nil {

        query = query.Where("deadline >= ?", \*dateFrom)

    } else if dateTo != nil {

        query = query.Where("deadline <= ?", \*dateTo)

    }

    // Zlicz niewykonane zadania

    if err := query.Count(&incompleteTaskCount).Error; err != nil {

        return 0, err

    }

    return incompleteTaskCount, nil

}

**user.go**

* **Model**: Reprezentuje użytkowników. Pola obejmują dane osobowe, adresowe, stanowisko, datę zatrudnienia.
* **Operacje**: Zarządzanie użytkownikami – tworzenie, edycja, wyszukiwanie.

package models

import (

    "time"

    "gorm.io/gorm"

)

type Position struct {

    gorm.Model

    Name string `gorm:"unique;not null"`

}

type User struct {

    gorm.Model

    Name             string `gorm:"not null"`

    Surname          string `gorm:"not null"`

    Email            string `gorm:"unique;not null"`

    PositionID       uint

    Position         Position  `gorm:"foreignKey:PositionID"`

    DateOfEmployment time.Time `gorm:"not null"`

    Phone            string    `gorm:"unique;not null"`

    Password         string    `gorm:"not null"`

    Street           string    `gorm:"not null"`

    City             string    `gorm:"not null"`

    State            string    `gorm:"not null"`

    Zip              string    `gorm:"not null"`

    Country          string    `gorm:"not null"`

    BankAccount      string    `gorm:"unique;not null"`

    NameBank         string    `gorm:"not null"`

}

func CreateUser(db \*gorm.DB, name string, surname string, email string, position Position, dateOfEmployment time.Time, phone, password, street, city, state, zip, country, bankAccount, nameBank string) (User, error) {

    user := User{

        Name:             name,

        Surname:          surname,

        Email:            email,

        Position:         position,

        DateOfEmployment: dateOfEmployment,

        Phone:            phone,

        Password:         password,

        Street:           street,

        City:             city,

        State:            state,

        Zip:              zip,

        Country:          country,

        BankAccount:      bankAccount,

        NameBank:         nameBank,

    }

    result := db.Create(&user)

    return user, result.Error

}

func CreatePosition(db \*gorm.DB, name string) (Position, error) {

    position := Position{

        Name: name,

    }

    result := db.Create(&position)

    return position, result.Error

}

func GetUserByEmail(db \*gorm.DB, email string) (User, error) {

    var user User

    err := db.Where("email = ?", email).First(&user).Error

    if err != nil {

        return user, err

    }

    return user, nil

}

func GetUserByID(db \*gorm.DB, id uint) (User, error) {

    var user User

    err := db.Preload("Position").Where("id = ?", id).First(&user).Order("id DESC").Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return User{}, gorm.ErrRecordNotFound

        }

        return User{}, err

    }

    return user, nil

}

func GetAllUsers(db \*gorm.DB) ([]User, error) {

    var users []User

    err := db.Find(&users).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return nil, gorm.ErrRecordNotFound

        }

        return nil, err

    }

    return users, nil

}

func GetAllWorkers(db \*gorm.DB) ([]User, error) {

    var users []User

    err := db.Joins("JOIN positions ON users.position\_id = positions.id").

        Where("positions.name = ?", "Magazynowy").

        Find(&users).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return nil, gorm.ErrRecordNotFound

        }

        return nil, err

    }

    return users, nil

}

func GetUserPositionByID(db \*gorm.DB, id uint) (name string, error error) {

    var position Position

    err := db.Model(&Position{}).

        Select("positions.name").

        Joins("JOIN users ON users.position\_id = positions.id").

        Where("users.id = ?", id).

        First(&position).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return "", nil

        }

        return "", err

    }

    return position.Name, nil

}

func GetUsersPassword(db \*gorm.DB, email string) (password string, error error) {

    var user User

    err := db.Model(&User{}).Select("password").Where("email = ?", email).First(&user).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return "", nil

        }

        return "", err

    }

    return user.Password, nil

}

func GetPositionByName(db \*gorm.DB, name string) (Position, error) {

    var position Position

    err := db.Where("name = ?", name).First(&position).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return Position{}, gorm.ErrRecordNotFound

        }

        return Position{}, err

    }

    return position, err

}

func CountUsersByPosition(db \*gorm.DB, dateFrom, dateTo \*time.Time) (map[string]int64, error) {

    var users []User

    // Zbuduj zapytanie z opcjonalnym filtrowaniem dat

    query := db.Preload("Position")

    if dateFrom != nil && dateTo != nil {

        query = query.Where("date\_of\_employment BETWEEN ? AND ?", \*dateFrom, \*dateTo)

    } else if dateFrom != nil {

        query = query.Where("date\_of\_employment >= ?", \*dateFrom)

    } else if dateTo != nil {

        query = query.Where("date\_of\_employment <= ?", \*dateTo)

    }

    // Pobierz użytkowników z bazy danych

    if err := query.Find(&users).Error; err != nil {

        return nil, err

    }

    // Licz użytkowników na podstawie pozycji

    var workerCount int64

    var hrCount int64

    for \_, user := range users {

        switch user.Position.Name {

        case "Magazynowy":

            workerCount++

        case "HR":

            hrCount++

        }

    }

    // Zwróć wynik

    return map[string]int64{

        "Magazynowy": workerCount,

        "HR":         hrCount,

    }, nil

}

**vacation.go**

* **Model**: Dotyczy wniosków urlopowych. Pola obejmują ID użytkownika, daty rozpoczęcia i zakończenia, status.
* **Operacje**: Składanie wniosków, akceptacja/odrzucenie, filtrowanie według statusu.

package models

import (

    "time"

    "gorm.io/gorm"

)

type Vacation struct {

    gorm.Model

    UserID          uint   `gorm:"not null"`

    User            User   `gorm:"foreignKey:UserID"`

    DateFrom        string `gorm:"not null"`

    DateTo          string `gorm:"not null"`

    DateCount       int    `gorm:"not null"`

    Status          string `gorm:"default:'Wysłany'"`

    RejectionReason string `gorm:"default:null"`

    Read            string `gorm:"default:'Nieodczytane'"` // 'Nieodczytane', 'Odczytane', lub null

}

func CreateVacation(db \*gorm.DB, userID uint, dateFrom, dateTo string, dateCount int, status string) (Vacation, error) {

    vacation := Vacation{

        UserID:    userID,

        DateFrom:  dateFrom,

        DateTo:    dateTo,

        DateCount: dateCount,

        Status:    status,

        Read:      "Nieodczytane", // Puste podczas tworzenia nowego urlopu

    }

    result := db.Create(&vacation)

    return vacation, result.Error

}

func MarkVacationAsRead(db \*gorm.DB, vacationID uint) error {

    return db.Model(&Vacation{}).Where("id = ?", vacationID).Update("read", "Odczytane").Error

}

func GetVacationByID(db \*gorm.DB, id uint) (Vacation, error) {

    var vacation Vacation

    err := db.Preload("User").Where("id = ?", id).First(&vacation).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return Vacation{}, gorm.ErrRecordNotFound

        }

        return Vacation{}, err

    }

    return vacation, nil

}

func GetVacationsByUserID(db \*gorm.DB, userID uint) ([]Vacation, error) {

    var vacations []Vacation

    err := db.Where("user\_id = ?", userID).Find(&vacations).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return nil, gorm.ErrRecordNotFound

        }

        return nil, err

    }

    return vacations, nil

}

func GetAllVacations(db \*gorm.DB) ([]Vacation, error) {

    var vacations []Vacation

    err := db.Where("status = ?", "Wysłany").Preload("User").Find(&vacations).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return nil, gorm.ErrRecordNotFound

        }

        return nil, err

    }

    return vacations, nil

}

func UpdateVacationDates(db \*gorm.DB, id uint, dateFrom, dateTo string, dateCount int) error {

    var vacation Vacation

    err := db.First(&vacation, id).Error

    if err != nil {

        if err == gorm.ErrRecordNotFound {

            return gorm.ErrRecordNotFound

        }

        return err

    }

    vacation.DateFrom = dateFrom

    vacation.DateTo = dateTo

    vacation.DateCount = dateCount

    return db.Save(&vacation).Error

}

func DeleteVacation(db \*gorm.DB, id uint) error {

    return db.Delete(&Vacation{}, id).Error

}

func UpdateVacationStatus(db \*gorm.DB, id uint, status string, rejectionReason \*string) error {

    updateData := map[string]interface{}{

        "status": status,

        "read":   "Nieodczytane", // Automatycznie ustaw na "Nieodczytane" przy zmianie statusu

    }

    if rejectionReason != nil {

        updateData["rejection\_reason"] = \*rejectionReason

    }

    return db.Model(&Vacation{}).Where("id = ?", id).Updates(updateData).Error

}

// Pobieranie liczby niewykonanych zadań dla użytkownika

func GetVacationCountByUserID(db \*gorm.DB, userID uint) (int64, error) {

    var count int64

    err := db.Model(&Vacation{}).Where("user\_id = ? AND read = ?", userID, "Nieodczytane").Count(&count).Error

    return count, err

}

func CountUsersOnVacation(db \*gorm.DB) (int64, error) {

    var count int64

    err := db.Model(&Vacation{}).

        Where("date\_from <= ? AND date\_to >= ?", time.Now().Format("2006-01-02"), time.Now().Format("2006-01-02")).

        Count(&count).Error

    if err != nil {

        return 0, err

    }

    return count, nil

}

**6.Aplikacja z widoku użytkownika**

**Obraz zawierający tekst, zrzut ekranu, design

Opis wygenerowany automatycznie**

Widok strony logowania

Obraz zawierający tekst, zrzut ekranu, numer, Czcionka

Opis wygenerowany automatycznie

Panel administracyjny HR

Obraz zawierający tekst, zrzut ekranu, Równolegle, oprogramowanie

Opis wygenerowany automatycznie

Wgląd w listę pracowników dla użytkownika HR w formie tabeli, oraz formularz umożliwiający dodanie nowego pracownika.

Obraz zawierający tekst, zrzut ekranu, numer, oprogramowanie

Opis wygenerowany automatycznie

Widok wszystkich zamówień realizowanych przez firmę, oraz formularz umożliwiający dodanie nowych zamówień.

Obraz zawierający tekst, zrzut ekranu, numer, Czcionka

Opis wygenerowany automatycznie

Widok zadań dla pracowników oraz formularz umożliwiający dodanie nowych zadań.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Widok wniosków urlopowych

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Widok powiadomienia informującego o nowej zmianie w zakładce urlop

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Strona internetowa

Opis wygenerowany automatycznie

Strona główna po zalogowaniu na konto użytkownika. Prezentuje funkcję umożliwiającą pomiar czasu pracy oraz bieżące powiadomienia.

Obraz zawierający tekst, zrzut ekranu, Czcionka

Opis wygenerowany automatycznie

Panel HR z poziomu użytkownika umożliwia dostęp do danych osobowych, adresu, danych bankowych oraz lokalizacji pracy. Pracownik ma również dostęp do zakładki Wnioski która umożliwia złożenie wniosku o urlop lub wygenerowania wniosku o zaświadczeniu pracy.

Obraz zawierający tekst, zrzut ekranu, Czcionka, design

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Czcionka, oprogramowanie

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, Czcionka, numer

Opis wygenerowany automatycznieObraz zawierający tekst, zrzut ekranu, Równolegle, oprogramowanie

Opis wygenerowany automatycznie

Obraz zawierający tekst, zrzut ekranu, diagram, oprogramowanie

Opis wygenerowany automatycznie

Po zalogowaniu się na konto admina, użytkownik ma dostęp do generowania raportów po uprzedni zastosowaniu filtru. Widzi również ilość zatrudnionych pracowników, zadań nieukończonych, łączną liczbę zamówień oraz liczbę osób przebywających na urlopie. Ma również wgląd do wykresów przedstawiających rodzaj oraz ilość faktur a także strukturę pracowników firmy.

**7. Podsumowanie**

Dokumentacja przedstawia kompleksowy opis systemu magazynowego dla sklepu rowerowego. Aplikacja została zaprojektowana z wykorzystaniem nowoczesnych technologii, takich jak Gin, GORM i Docker, aby zapewnić wydajność, skalowalność oraz intuicyjną obsługę. System integruje funkcjonalności umożliwiające efektywne zarządzanie magazynem, zasobami ludzkimi i zadaniami pracowników.