Dokumentacja projektu

System zarządzania zadaniami z wykorzystaniem API

20.01.2024 r.

Piotr Gwiazda

1. Temat projektu

   Tematem projektu było napisanie aplikacji w technologii ASP.NET Core WebAPI. Do wykonania zostały użyte języki: C#, Microsoft Server SQL i React.

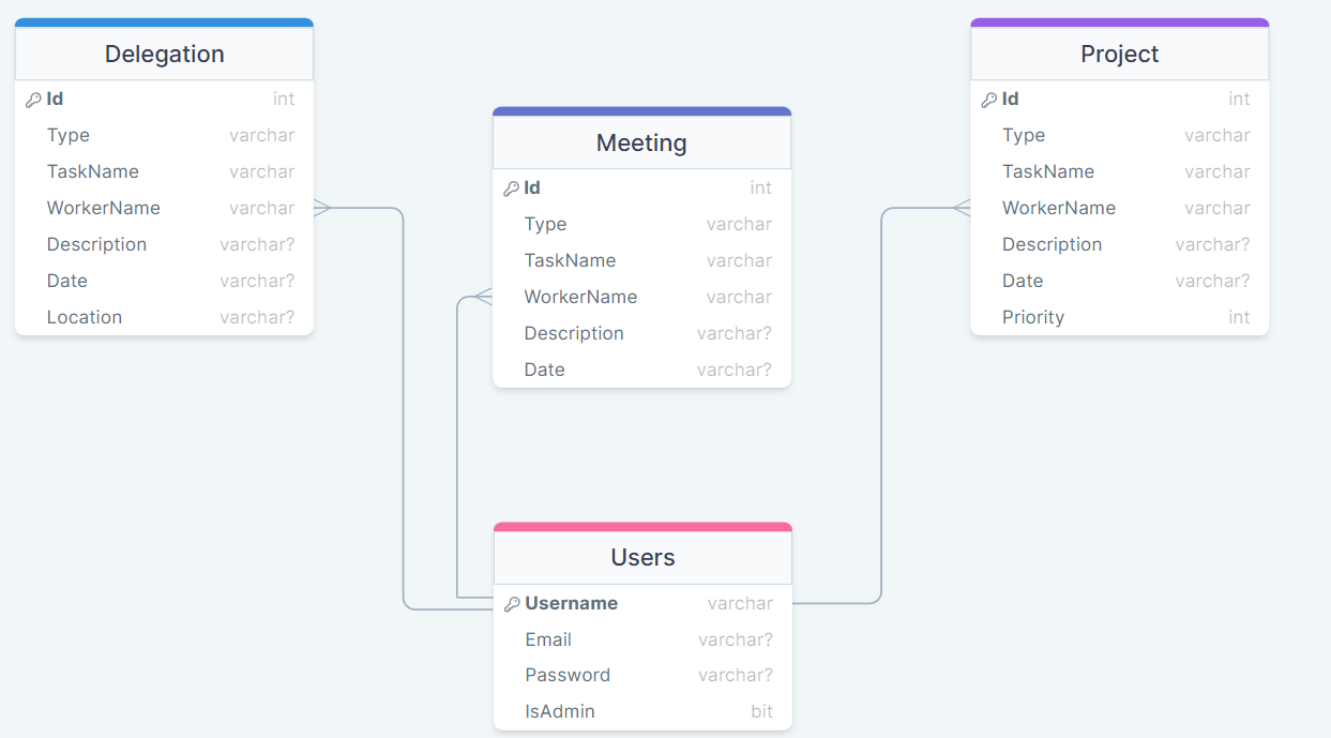
2. Funkcjonalność aplikacji

   - Użytkownik może się zarejestrować i zalogować jako menadżer lub pracownik
   - Menadżer ma możliwość dodawania i usuwania pracowników oraz przypisywania, zmieniania i usuwania im zadań
   - Menadżer i pracownicy mogą wyświetlić swoje zadania w kalendarzu
   - Użytkownicy może zmienić konto po wylogowaniu się

3. Baza danych

   Została stworzona w SQL, do niej są wysyłane i pobierane dane.

   Diagram ERD w bazie danych:



   Zapytania do bazy danych:

   - Pobieranie użytkowników

     ```
     SELECT * FROM USERSTABLE
     ```

   - Dodawanie użytkownika

     ```
     INSERT INTO USERSTABLE (Email, Username, Password, IsAdmin, Info)
     VALUES ('{user.Email}', '{user.Username}', '{user.Password}', '{val}', '{user.SetInfo()}')
     ```

   - Usuwanie użytkownika

     ```
     DELETE FROM USERSTABLE WHERE Username = '{username}'
     ```

   - Edytowanie użytkownika

     ```
     UPDATE USERSTABLE SET Email = '{user.Email}', Password = '{user.Password}', WHERE Username = '{user.Username}'
     ```

- Pobieranie projektów

```
'SELECT * FROM PROJECT
```

- Dodawanie projektu

```
INSERT INTO PROJECT (Id, Type, WorkerName, TaskName, Description, Date, Priority)
VALUES ('{proj.Id}', '{proj.Type}', '{proj.WorkerName}', '{proj.Name}', '{proj.ViewTask()}', '{proj.Date}', '{proj.Priority}')
```

- Usuwanie projektu

```
DELETE FROM PROJECT WHERE Id = {id}
```

- Edytowanie projektu

```
UPDATE PROJECT SET WorkerName = '{proj.WorkerName}', TaskName = '{proj.Name}', Description = '{proj.ViewTask()}',
Date = '{proj.Date}', Priority = '{proj.Priority}' WHERE Id = {proj.Id}
```

- Pobieranie spotkań

```
SELECT * FROM MEETING
```

- Dodawanie spotkania

```
INSERT INTO MEETING (Id, Type, WorkerName, TaskName, Description, Date) VALUES
VALUES ('{meet.Id}', '{meet.Type}', '{meet.WorkerName}', '{meet.Name}', '{meet.ViewTask()}', '{meet.Date}')
```

- Usuwanie spotkania

```
DELETE FROM MEETING WHERE Id = {id}
```

- Edytowanie spotkania

```
UPDATE MEETING SET WorkerName = '{meet.WorkerName}', TaskName = '{meet.Name}', Description = '{meet.ViewTask()}',
Date = '{meet.Date}' WHERE Id = {meet.Id}
```

- Pobieranie delegacji

```
SELECT * FROM DELEGATION
```

- Dodawanie delegacji

```
INSERT INTO DELEGATION (Id, Type, WorkerName, TaskName, Description, Date, Location) VALUES
VALUES ('{deleg.Id}', '{deleg.Type}', '{deleg.WorkerName}', '{deleg.Name}', '{deleg.ViewTask()}', '{deleg.Date}', '{deleg.Location}')
```

- Usuwanie delegacji

```
DELETE FROM DELEGATION WHERE Id = {id}
```

- Edytowanie delegacji

```
UPDATE DELEGATION SET WorkerName = '{deleg.WorkerName}', TaskName = '{deleg.Name}', Description = '{deleg.ViewTask()}',
Date = '{deleg.Date}', Location = '{deleg.Location}' WHERE Id = {deleg.Id}
```

Powyższe zapytania są wywoływane w klasie Database. Dla dodawania, edytowania i usuwania używana jest metoda Execute.

```
public static void Execute(string command)
{
    using (SqlConnection conn = new SqlConnection(@"Data Source=(local); Initial Catalog=Users; User ID=root; Password=root;"))
    {
        using (SqlCommand cmd = new SqlCommand(command, conn))
        {
            conn.Open();
            cmd.ExecuteScalar();
            conn.Close();
        }
    }
}
```

Zapytanie jest pobierane jako string i wywoływane przez ExecuteScalar.

Dla pobierania istnieją dwie metody, które zwracają listy użytkowników i zadań.

Dla użytkowników:

```
public static List<User> GetUserData() {
    List<User> list = new List<User>();
    using (SqlConnection conn = new SqlConnection(@"Data Source=(local); Initial Catalog=Users; User ID=root; Password=root;"))
    {
        using (SqlCommand cmd = new SqlCommand("SELECT * FROM USERSTABLE", conn))
        {
            conn.Open();
            using (SqlDataReader reader = cmd.ExecuteReader())
            {
                int emailOrdinal = reader.GetOrdinal("Email");
                int usernameOrdinal = reader.GetOrdinal("Username");
                int passwordOrdinal = reader.GetOrdinal("Password");
                int isAdminOrdinal = reader.GetOrdinal("IsAdmin");
                int infoOrdinal = reader.GetOrdinal("Info");
                while(reader.Read())
                {
                    User user = new User();
                    user.Email = reader.GetString(emailOrdinal);
                    user.Username = reader.GetString(usernameOrdinal);
                    user.Password = reader.GetString(passwordOrdinal);
                    user.IsAdmin = reader.GetBoolean(isAdminOrdinal);
                    user.Info = reader.GetString(infoOrdinal);
                    list.Add(user);
                }
            }
            conn.Close();
        }
    }
    return list;
}
```

Dla zadań:

```
public static List<Task> GetTaskData() {
    List<Task> list = new List<Task>();
    using (SqlConnection conn = new SqlConnection(@"Data Source=(local); Initial Catalog=Users; User ID=root; Password=root;"))
    {
        using (SqlCommand cmd = new SqlCommand("SELECT * FROM PROJECT", conn))
        {
            conn.Open();
            using (SqlDataReader reader = cmd.ExecuteReader())
            {
                int idOrdinal = reader.GetOrdinal("Id");
                int taskNameOrdinal = reader.GetOrdinal("TaskName");
                int workerNameOrdinal = reader.GetOrdinal("WorkerName");
                int taskInfoOrdinal = reader.GetOrdinal("Description");
                int dateOrdinal = reader.GetOrdinal("Date");
                int priorityOrdinal = reader.GetOrdinal("Priority");
                while(reader.Read())
                {
                    var task = new ProjectTask();
                    task.Id = reader.GetInt32(idOrdinal);
                    task.Name = reader.GetString(taskNameOrdinal);
                    task.WorkerName = reader.GetString(workerNameOrdinal);
                    task.TaskInfo = reader.GetString(taskInfoOrdinal);
                    task.Date = reader.GetString(dateOrdinal);
                    task.Priority = reader.GetInt32(priorityOrdinal);
                    list.Add(task);
                }
            }
            conn.Close();
        }
```

```
using (SqlCommand cmd = new SqlCommand("SELECT * FROM MEETING", conn))
{
    conn.Open();
    using (SqlDataReader reader = cmd.ExecuteReader())
    {
        int idOrdinal = reader.GetOrdinal("Id");
        int taskNameOrdinal = reader.GetOrdinal("TaskName");
        int workerNameOrdinal = reader.GetOrdinal("WorkerName");
        int taskInfoOrdinal = reader.GetOrdinal("Description");
        int dateOrdinal = reader.GetOrdinal("Date");
        while(reader.Read())
        {
            var task = new MeetingTask();
            task.Id = reader.GetInt32(idOrdinal);
            task.Name = reader.GetString(taskNameOrdinal);
            task.WorkerName = reader.GetString(workerNameOrdinal);
            task.TaskInfo = reader.GetString(taskInfoOrdinal);
            task.Date = reader.GetString(dateOrdinal);
            list.Add(task);
        }
    }
    conn.Close();
}
```

```
    using (SqlCommand cmd = new SqlCommand("SELECT * FROM DELEGATION", conn))
    {
        conn.Open();
        using (SqlDataReader reader = cmd.ExecuteReader())
        {
            int idOrdinal = reader.GetOrdinal("Id");
            int taskNameOrdinal = reader.GetOrdinal("TaskName");
            int workerNameOrdinal = reader.GetOrdinal("WorkerName");
            int taskInfoOrdinal = reader.GetOrdinal("Description");
            int dateOrdinal = reader.GetOrdinal("Date");
            int locationOrdinal = reader.GetOrdinal("Location");
            while(reader.Read())
            {
                var task = new DelegationTask();
                task.Id = reader.GetInt32(idOrdinal);
                task.Name = reader.GetString(taskNameOrdinal);
                task.WorkerName = reader.GetString(workerNameOrdinal);
                task.TaskInfo = reader.GetString(taskInfoOrdinal);
                task.Date = reader.GetString(dateOrdinal);
                task.Location = reader.GetString(locationOrdinal);
                list.Add(task);
            }
        }
        conn.Close();
    }
}
return list;
```

Te metody pobierają indeksy kolumn, z których poźniej są pobierane dane i przekazywane do listy, która jest zwracana przez metody.

4. API

Zostało napisane przy pomocy języka C#. Umożliwia komunikację między bazą danych i stroną internetową. Jest podzielone na klasy Controller, Service i Model dla zadań (Task) i użytkowników (User) oraz klasę Controller dla wytwarzania JSON Web Token (JWT). Service wywołuje metody z bazy danych.

Użytkownicy:

- Model:

```
public class User
{
    3 references
    public string? Email {get; set;}
    11 references
    public string? Username {get; set;}
    5 references
    public string? Password {get; set;}
    5 references
    public bool IsAdmin { get; set; }
    1 reference
    public string? Info {get; set;}
    1 reference
    public User()
    {
    }
    1 reference
    public string SetInfo(){
        string info = IsAdmin ? $"{Username} is the Manager. He assigns tasks.":$"{Username} is the Employee. He completes tasks.";
        return info;
    }
}
```

Zawiera pola, oraz metodę string SetInfo, która ustawia pole Info w bazie danych

- Service:

```
public class UserService
{
    4 references
    static List<User> Users { get; set; }
    0 references
    static UserService()
    {
        Users = Database.GetUserData();
    }
    1 reference
    public static List<User> GetAll() {
        Users = Database.GetUserData();
        return Users;
    }
```

UserService zawiera statyczną listę Users, która jest inicjowana w konstruktorze, pobierając z użytkowników z bazy danych. Metoda GetAll aktualizuje listę i zwraca ją.

```
public static User? Get(string username)
{
    return Users.FirstOrDefault(u => u.Username == username);
}
```

Metoda Get zwraca obiekt User, któremu odpowiada przekazany string username.

```
public static void Add(User user)
{
    Database.Execute(Database.InsertUser(user));
}
```

Metoda Add wywołuje zapytanie o utworzenie użytkownika do bazy danych.

```
public static void Delete(string username)
{
    Database.Execute(Database.DeleteUser(username));
}
```

Metoda Delete wywołuje zapytanie o usunięcie użytkownika do bazy danych.

```
public static void Update(User user)
{
    Database.Execute(Database.UpdateUser(user));
}
```

Metoda Update wywołuje zapytanie o aktualizacji użytkownika do bazy danych.

- Controller:

```
[ApiController]
[Route("[controller]")]
0 references
public class UserController:ControllerBase
{

    [HttpGet]
    0 references
    public ActionResult<List<User>> GetAll() =>
    UserService.GetAll();
    [HttpGet("{username}")]
    1 reference
    public ActionResult<User> Get(string username)
    {
        var user = UserService.Get(username);

        if(user == null)
            return NotFound();

        return user;
    }
```

Metoda GetAll obsługuje żądanie HTTP GET i wywołuje metodę GetAll z UserService i zwraca listę użytkowników jako rezultat. Metoda Get obsługuje żądanie HTTP GET dla konkretnego użytkownika na podstawie jego nazwy. Jeżeli użytkownik nie istnieje zwraca kod HTTP 404, w przeciwnym razie zwraca użytkownika.

```
[HttpPost]
0 references
public IActionResult Create(User user)
{
    var existingUser = UserService.Get(user.Username+"");
    if(existingUser != null)
    {
        UserService.Update(user);
        return NoContent();
    }
    UserService.Add(user);
    return CreatedAtAction(nameof(Get), new { username = user.Username}, user);
}
```

Metoda Create obsługuje żądanie HTTP POST. Sprawdza czy istnieje użytkownik przy pomocy metody Get z UserService. Jeżeli istnieje, aktualizuje informacje o użytkowniku i zwraca kod HTTP 204 No Content. Jeżeli nie istnieje, zostaje dodany za pomocą metody Add z UserService i kod HTTP 201 Created wraz z utworzonym użytkownikiem.

```
[HttpPut("{username}")]
0 references
public IActionResult Update(string username, User user)
{
    if (username != user.Username)
        return  BadRequest();
    var existingUser = UserService.Get(username);
    if (existingUser is null)
        return NotFound();

    UserService.Update(user);

    return NoContent();
}
```

Metoda Update obsługuje żądanie HTTP PUT dla użytkownika, któremu odpowiada podany string username. Przyjmowane są dwa parametry string username i obiekt User. Jeżeli nazwa przyjęta użytkownika nie zgadza się z nazwą użytkownika w przesłanym obiekcie zwracany jest kod HTTP 400 Bad Request. Następnie metoda sprawdza czy istnieje użytkownik o podajnej nazwie, gdy nie istnieje zwraca kod HTTP 404 Not Found. Jeżeli istnieje, aktualizuje użytkownika przy pomocy metody Update z UserService i zwraca kod HTTP 204 No Content.

```
[HttpDelete("{username}")]
0 references
public IActionResult Delete(string username)
{
    var user = UserService.Get(username);

    if (user is null)
        return NotFound();

    UserService.Delete(username);

    return NoContent();
}
```

Metoda Delete obsługuje żadanie HTTP DELETE dla użytkownika, któremu odpowiada podany string username. Przy pomocy metody Get z UserService sprawdza czy istnieje użytkownik o przyjętym username. Jeżeli nie istnieje zwraca kod HTTP 404 Not Found. Natomiast gdy istnieje zostaje on usunięty za pomocą metody Delete z UserService i zwraca kod HTTP 204 No Content.

Zadania:

- Modele:

```
public abstract class Task
{
    13 references
    public int Id {get; set;}
    3 references
    public int Discriminator {get; set;}
    12 references
    public string? Name {get; set;}
    13 references
    public string? Type {get; set;}
    9 references
    public string? WorkerName {get; set;}
    12 references
    public string? Date {get; set;}
    3 references
    public string? TaskInfo {get; set;}
    7 references
    public abstract string ViewTask();
}
```

Abstrakcyjna klasa Task. Pola Discriminator i Type służą do rozróżnienia zadań dziedziczących po niej. Metoda ViewTask ma zwracać informacje na temat zadania.

```
public enum TypesOfTasks
{
    1 reference
    Project,
    1 reference
    Meeting,
    1 reference
    Delegation
}
```

Enumerator TypesOfTasks definiuje wartości odpowiadające klasom dziedziczącym po Task. Służy do rozróżniania klas potomnych klasy abstrakcyjnej Task.

```
builder.Services.AddControllers();
builder.Services.AddControllers().AddNewtonsoftJson(options =>
{

    options.SerializerSettings.Converters.Add(
    JsonSubtypesConverterBuilder
    .Of(typeof(Task),"Discriminator")
    .RegisterSubtype(typeof(ProjectTask), TypesOfTasks.Project)
    .RegisterSubtype(typeof(MeetingTask), TypesOfTasks.Meeting)
    .RegisterSubtype(typeof(DelegationTask), TypesOfTasks.Delegation)
    .SerializeDiscriminatorProperty()
    .Build()
    );
        });
```

Dodanie powyższego kontrolera w klasie Program jest wymagane, żeby API rozróżniało klasy potemne klasy abstrakcyjnej Task.

```
public class ProjectTask:Task
{
    4 references
    public int Priority {get; set;}
    1 reference
    public ProjectTask()
    {
        Type = "Project";
        Discriminator = 0;
    }
    7 references
    public override string ViewTask()
    {
        string s;
        switch(Priority){
            case 1:
                s = "low priority";
                break;
            case 2:
                s = "medium priority";
                break;
            default:
                s = "high priority";
                break;
        }
        return $"{Type} {Name} is {s} and needs to be completed before {Date}.";
    }
}
```

Klasa ProjectTask dziedzicząca po Task z dodatkowym polem int Priority. W konstruktorze ustalane są wartości pól Type i Discriminator.

```
public class MeetingTask:Task
{
    1 reference
    public MeetingTask()
    {
        Type = "Meeting";
        Discriminator = 1;
    }
    7 references
    public override string ViewTask()
    {
        return $"{Type} about {Name} on {Date}.";
    }
}
```

Klasa MeetingTask dziedzicząca po Task. W konstruktorze ustalane są wartości pól Type i Discriminator.

```
public class DelegationTask:Task
{
    4 references
    public string? Location {get; set;}
    1 reference
    public DelegationTask()
    {
        Type = "Delegation";
        Discriminator = 2;
    }
    7 references
    public override string ViewTask()
    {
        return $"{Type} {Name} to {Location} on {Date}.";
    }
}
```

Klasa DelegationTask dziedzicząca po Task z dodatkowym polem string Location. W konstruktorze ustalane są wartości pól Type i Discriminator.

- Service:

```
public class TaskService
{
    4 references
    static List<Task> Tasks {get; set;}
    3 references
    static int nextId;
    0 references
    static TaskService()
    {
        Tasks = Database.GetTaskData();
        if(Database.GetTaskData().Count!=0)
        {
            nextId = Database.GetTaskData().Max(t => t.Id);
        }
    }
     1 reference
     public static List<Task> GetAll() {
        Tasks = Database.GetTaskData();
        return Tasks;
    }
    3 references
    public static Task? Get(int id)
    {
        return Tasks.FirstOrDefault(t => t.Id == id);
    }

    1 reference
    public static void Add(Task task)
    {

        nextId++;
        task.Id = nextId;
        Database.Execute(Database.InsertTask(task));
    }
```

```
public static void Delete(int id)
{
    Database.Execute(Database.DeleteTask(id));
}

1 reference
public static void Update(Task task)
{
    Database.Execute(Database.UpdateTask(task));
}
```

TaskService działa analogicznie do UserService z dodatkiem statycznego pola int nextId, które przechowuje informacje o kluczu ostatniego zadania. To pole zmienia się wraz z dodaniem kolejnego elementu do listy.

- Controller:

```
[ApiController]
[Route("[controller]")]
0 references
public class TaskController:ControllerBase
{

    [HttpGet]
    0 references
    public ActionResult<List<Task>> GetAll() =>
    TaskService.GetAll();
    [HttpGet("{id}")]
    1 reference
    public ActionResult<Task> Get(int id)
    {
        var task = TaskService.Get(id);

        if(task == null)
            return NotFound();

        return task;
    }

    [HttpPost]
    0 references
    public IActionResult Create(Task task)
    {
        TaskService.Add(task);
        return CreatedAtAction(nameof(Get), new { id = task.Id}, task);
    }
```

```
[HttpPut("{id}")]
0 references
public IActionResult Update(int id, Task task)
{
    if (id != task.Id)
    {
        return  BadRequest();
    }
    var existingTask = TaskService.Get(id);
    if (existingTask is null)
        return NotFound();

    TaskService.Update(task);

    return NoContent();
}
```

```
[HttpDelete("{id}")]
0 references
public IActionResult Delete(int id)
{
    var task = TaskService.Get(id);
    if (task is null)
        return NotFound();

    TaskService.Delete(id);

    return NoContent();
}
```

Działa analogicznie do kontrolera użytkownika. Zamiast przyjmować username w metodach przyjmuje id.

- LoginController (tworzy JWT):

```
[ApiController]
[Route("[Controller]")]
0 references
public class LoginController: ControllerBase
{
    4 references
    private IConfiguration _config;
    0 references
    public LoginController(IConfiguration configuration)
    {
        _config = configuration;
    }
    1 reference
    private User? AuthenticateUser(User user)
    {
        if(Database.GetUserData().Exists(u => u.Username == user.Username && u.Password == user.Password && u.IsAdmin == user.IsAdmin))
        {
            return user;
        }
        return null;
    }
```

W konstruktorze przekazywany jest obiekt IConfiguration, który pozwala na dostęp do konfiguracji aplikacji, w tym do JWT. Sprawdza, czy użytkownik o podanym username, password i zmiennej typu boolean IsAdmin istnieje w bazie danych (Database.GetUserData()). Jeżeli tak, zwraca obiekt User. W przeciwnym razie zwraca null.

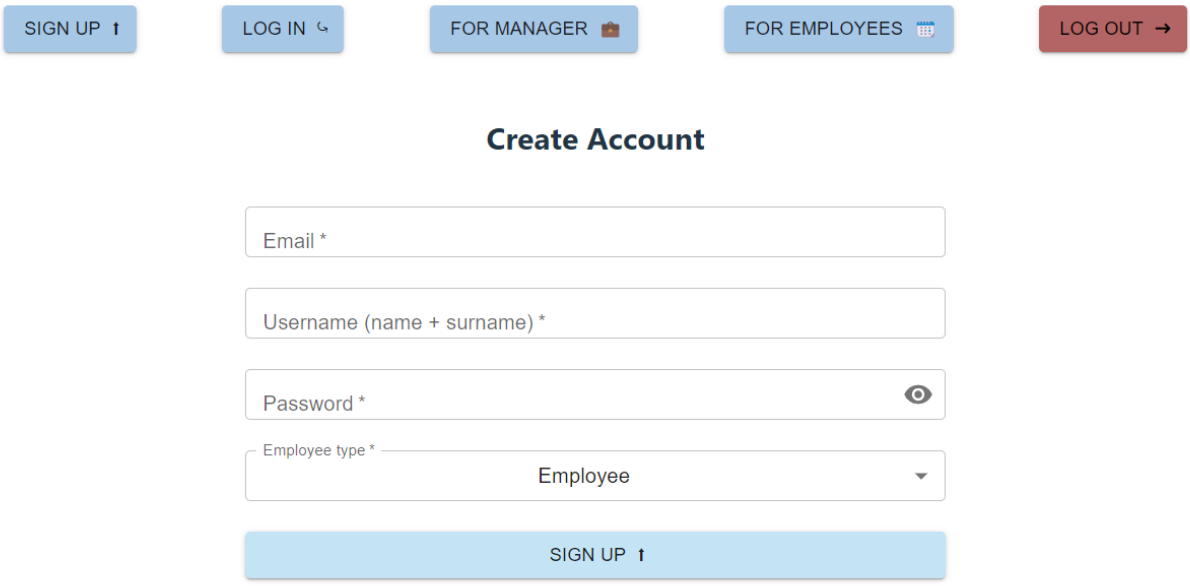
```
private string GenerateToken(User user)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(_config["Jwt:Issuer"], _config["Jwt:Audience"], null,
    expires: DateTime.Now.AddHours(1), signingCredentials: credentials);
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Metoda GenerateToken generuje token JWT dla danego użytkownika. SymmetricSecurityKey używane jest do zdefiniowania klucza szyfrującego. SigningCredentials określa sposoby uwierzytelniania podczas tworzenia tokena. JwtSecurityToken tworzy token JWT z określonymi parametrami. JwtSecurityTokenHandler().WriteToken(token) przekształca token JWT na jego reprezentację tekstową.

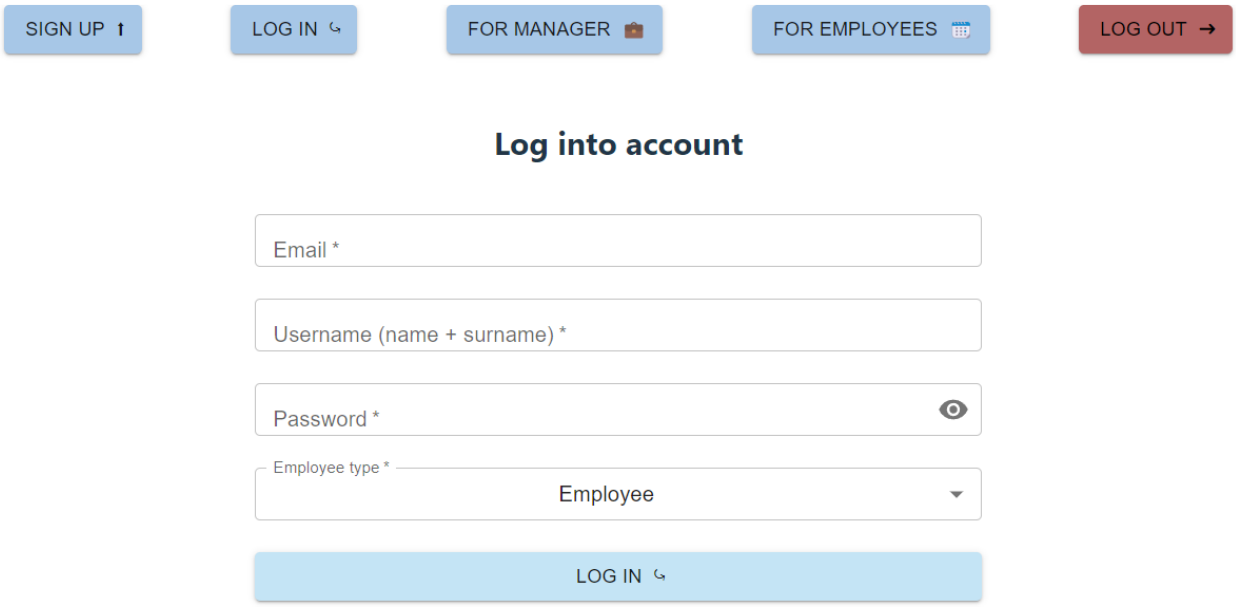
```
[AllowAnonymous]
[HttpPost]
0 references
public IActionResult Login(User user)
{
    IActionResult result = Unauthorized();
    var user_ = AuthenticateUser(user);
    if (user_ != null)
    {
        var token = GenerateToken(user_);
        result = Ok(new {token = token});
    }
    return result;
}
```

Metoda Login obsługuje żądanie HTTP POST. AllowAnonymous pozwala na dostęp do tej metody nieuwierzytelnionym użytkownikom. Domyślnie zwraca kod HTTP 401 Unauthorized, jeżeli autoryzacja powiedzie się za pomocą metod AuthenticateUser, zostanie stworzony token i przekazany w odpowiedzi wraz z kodem HTTP 200 OK.

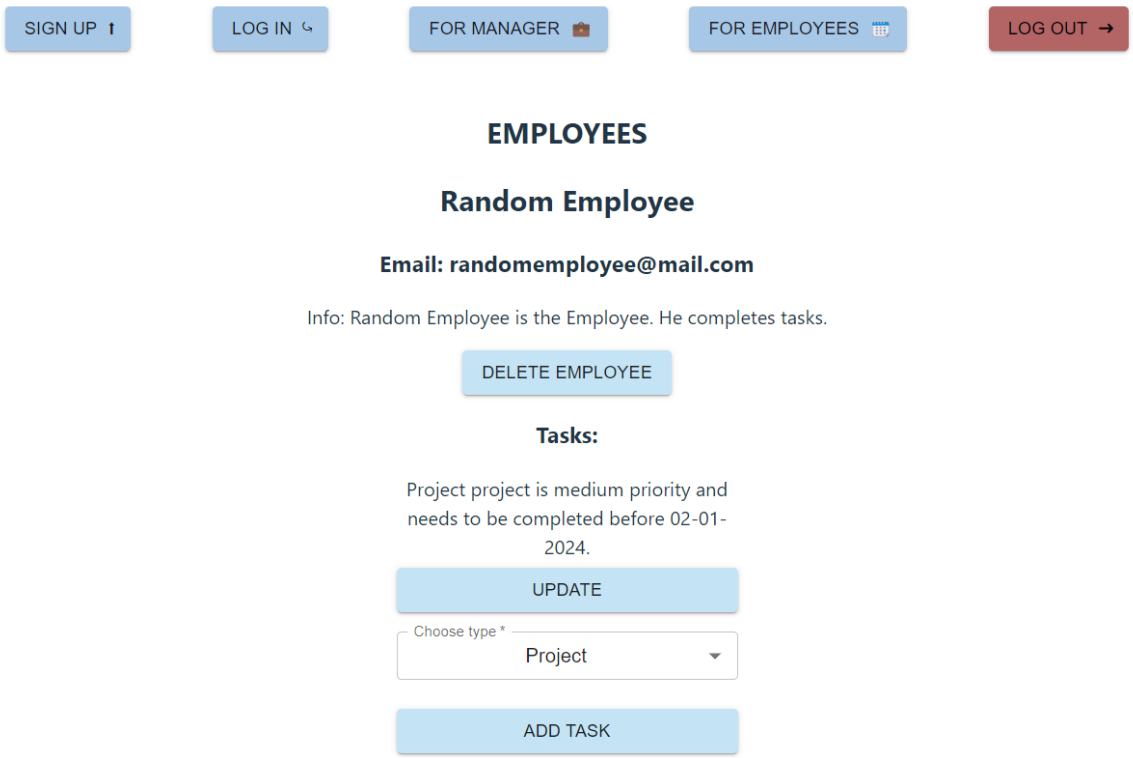
5. Strona internetowa

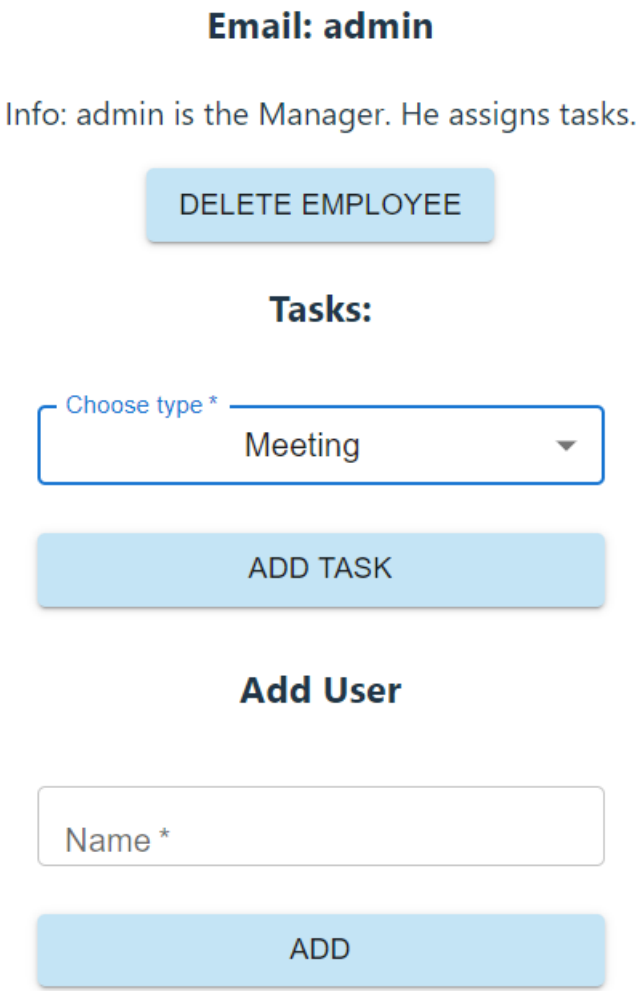


Strona startowa służąca do zarejestrowania użytkownika. Przyciski na górze strony przekierowują na odpowiednie strony, jeżeli użytkownik się zarejestrował. Po wypełnieniu formularza konto użytkownika zostanie stworzone i użytkownik zostanie przekierowany na stronę do logowania. Na tą stronę przekierowuje przycisk SIGN UP. Przycisk z napisem LOG OUT wyloguje użytkownika, jeżeli jest zalogowany.

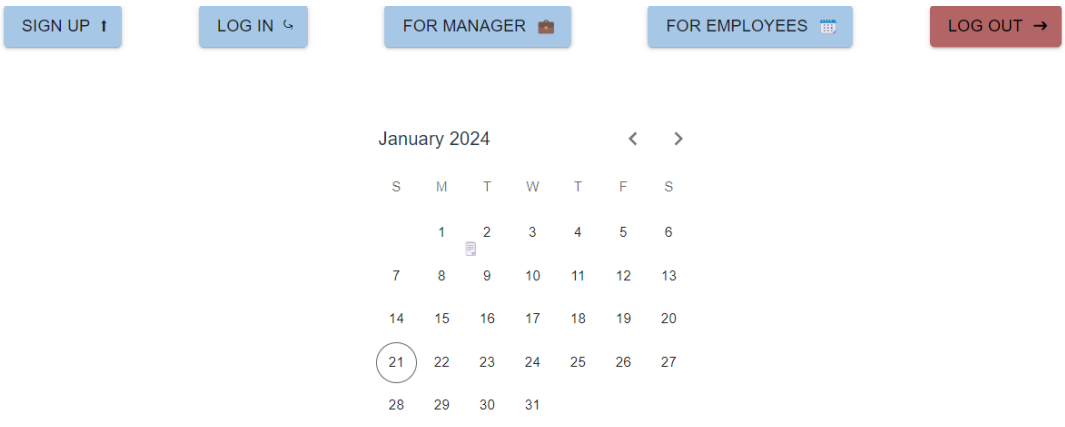


Strona służąca za zalogowanie użytkownika. Po wypełnieniu formularza użytkownik zostanie zalogowany. Na tą stronę przekierowuje przycisk LOG IN.

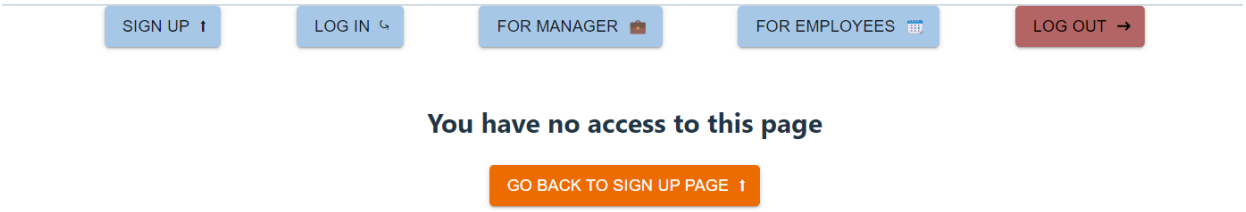




Strona dla menadżera. Służy do dodawania i usuwania użytkowników oraz dodania, usuwania i zmieniania zadań dodanym użytkownikom. Przyciski z napisem DELETE usuwają użytkownika lub zadanie. Przyciski z napisem ADD dodają zadania lub pracowników i przycisk UPDATE zmienia zadanie odpowiedniemu użytkownikowi po wypełnieniu formularza. Przycisk FOR MANAGER przekierowuje na tą stronę.



Strona dla pracowników. Na niej znajduje się kalendarz i informacje o zadaniach konkretnego użytkownika. Przycisk FOR EMPLOYEES przekierowuje na tą stronę.



Strona, która wyświetli się gdy niezalogowany użytkownik kliknie w przycisk FOR MANAGER lub FOR EMPLOYEE.

6. Instalacja

Program wymaga zainstalowania Microsoft Server SQL oraz ustawienia portu localhost API na 7113 i portu localhost Reacta na 5173. Do uruchomienia należy wpisać komendy:

- sqlcmd -s mssqlserver -e
    i. create database users
    ii. go
- dotnet run – w folderze Api
- npm run dev – w /Api/page

Następnie wejść na stronę localhost:5173.