Integrowanie aplikacji .NET z systemami baz danych

Dokumentacja projektu

„Sklep internetowy ze sprzętem narciarskim”

Bartłomiej Mazurkiewicz

173670, L02, 3EF-DI/AA

Spis treści

Założenia projektu

Projekt zakładał stworzenie API w języku C# na platformie .NET dla sklepu z artykułem narciarskim. Zostały stworzone takie klasy jak: User, Announcement, Review, SavedAnnouncements, ShoppingCart, Orders. Każda z klas zawiera metody CRUD używane w kontrolerach.

Repozytorium: https://github.com/mazurek321/Backend_Store

Uruchomienie projektu

1. **Sklonowanie repozytorium, komenda:** git clone https://github.com/mazurek321/Backend_Store.git
2. **Wejście do folderu, komenda**: cd Backend_Store/
3. **Uruchomić aplikację Docker Desktop.**
4. **W folderze "Backend_Store" komenda:** docker build .
5. **W tej samej lokalizacji komenda:** docker compose up
6. **W tej samej lokalizacj i nowym terminalu komenda:** dotnet ef migrations add „NowaMigracja"
7. **Aktualizacja bazy:** dotnet ef database update
8. **Dodanie funkcji składowanej w bazie danych z pliku txt.**
9. **Uruchomienie projektu:** dotnet run
10. **Wejście do swaggera, w przeglądarce internetowej:** http://localhost:5050/swagger/index.html

Logowanie na konto admina:

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

"password": "admin"

Kod programu

Program zawiera klasy takie jak: User, Announcement, Item, ShoppingCart, ShoppingCartItem, SavedAnnouncements, Reviews, Orders, OrderedItems.

Każda z klas posiada deklarację zmiennych, konstruktor oraz metody służące do ustawiania poszczególnych wartości, a także ich aktualizowanie.

Jedną ze wspomnianych klas jest klasa User, która reprezentuje użytkownika.

Klasa User

```
namespace projekt.src.Models.Users;

public class User
{
    public User(){}

    private User(
        UserId id, Email email, Name name, LastName lastname, Password password,
        Address? address, Address? location, PostCode? postcode, Phone? phone, AccountType role, DateTime createdAt
    ){
        Id = id;
        Email = email;
        Name = name;
        LastName = lastname;
        Password = password;
        Address = address;
        Location = location;
        PostCode = postcode;
        Phone = phone;
        Role = role;
        CreatedAt = createdAt;
    }

    public UserId Id {get; private set; }
    public Email Email { get; private set; }
    public Name Name {get; private set; }
    public LastName LastName {get; private set; }
    public Password Password {get; private set; }
    public Address? Address { get; private set; }
    public Address? Location { get; private set; }
    public PostCode? PostCode {get; private set;}
    public Phone? Phone { get; private set; }
    public AccountType Role { get; private set; }
    public DateTime CreatedAt{ get; private set; }
```

Rysunek 1 Zmienne klasy User oraz konstruktor

Przedstawione na rysunku 1 zmienne mają własnoręcznie stworzone typy, np. Id użytkownika jest typu UserId, Email typu Email. Występują także zmienne opcjonalne, takie jak Address, Location, PostCode czy też Phone. Ważną zmienną jest tutaj Role typu AccountType, ponieważ określa ona typ konta. W zależności od jego typu, możliwe są odpowiednie funkcjonalności.

```
public static User NewUser(
    Email email, Name name, LastName lastname, Password password,
    Address? address, Address? location, PostCode? postcode, Phone? phone, DateTime createdAt
){
    var id = new UserId(Guid.NewGuid());
    return new User(id, email, name, lastname, password, address, location, postcode, phone, AccountType.User(), createdAt);
}

public static User NewAdmin(
    Email email, Name name, LastName lastname, Password password,
    Address? address, Address? location, PostCode? postcode, Phone? phone, DateTime createdAt
){
    var id = new UserId(Guid.NewGuid());
    return new User(id, email, name, lastname, password, address, location, postcode, phone, AccountType.Admin(), createdAt);
}

public void UpdatePassword(Password newPassword)
{
    Password = newPassword;
}

public void UpdateInformation(Name? name, LastName? lastName, Address? address, Address? location, PostCode? postCode, Phone? phone)
{
    Name = name;
    LastName = lastName;
    Address = address;
    Location = location;
    PostCode = postCode;
    Phone = phone;
}
```

Rysunek 2 Metody klasy User

Na rysunku 2 pokazane są metody klasy User, służące do tworzenia nowego użytkownika lub administratora, aktualizowania hasła czy też osobna metoda do aktualizacji pozostałych informacji.

Typy zmiennych klasy User

```
using projekt.src.Exceptions;

namespace projekt.src.Models.Users;

public record UserId
{
    public UserId(Guid value){
        if(value == Guid.Empty) throw new CustomException("Invalid id.");
        Value = value;
    }

    public static implicit operator UserId (string value){
        if(string.IsNullOrEmpty(value)) throw new CustomException("Invalid id.");
        return new UserId(Guid.Parse(value));
    }

    public Guid Value {get; }
}
```

Rysunek 3 Typ UserId

Na rysunku 3 pokazany jest typ UserId, który jest wykorzystywany dla Id użytkownika. Występuje on jako record, gdzie przy inicjalizowaniu zmiennej Id najpierw jest wykonywana procedura zawarta w tym rekordzie. Sprawdzane jest tutaj, czy wartość nie jest pusta. Jeśli nie, to wartość jest przypisywana do zmiennej Value.

```
using System.Text.RegularExpressions;
using projekt.src.Exceptions;

namespace projekt.src.Models.Users;

public record Email
{
    public Email(string value){
        if(string.IsNullOrEmpty(value) || !checkValidation(value)) throw new CustomException("Invalid email.");
        Value = value;
    }

    public string Value {get; }

    private bool checkValidation(string value){
        return Regex.IsMatch(value,
            @"^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$" );
    }
}
```

Rysunek 4 Typ Email

Dla typu Email zastosowano dodatkowo walidację sprawdzającą poprawność zapisu emaila. Jeśli nie zostanie wykryty żaden błąd, wartość zostanie przypisana do zmiennej.

```
using System.Security.Cryptography;
using System.Text;
using projekt.src.Exceptions;

namespace projekt.src.Models.Users;

public record Password
{
    public Password(string value){
        if(string.IsNullOrWhiteSpace(value) || value.Length is > 200 or < 5) throw new CustomException("Invalid id.");
        Value = value;
    }

    public string Value {get; }

    public string CalculateMD5Hash(string input)
    {
        MD5 md5 = MD5.Create();
        byte[] inputBytes = Encoding.UTF8.GetBytes(input);
        byte[] hash = md5.ComputeHash(inputBytes);

        StringBuilder sb = new StringBuilder();
        for(int i=0; i< hash.Length; i++)
        {
            sb.Append(hash[i].ToString("X2"));
        }
        return sb.ToString();
    }
}
```

Rysunek 5 Typ Password

Dla typu Password sprawdzana jest długość hasła. Jeśli jest ona dłuższa niż 200 znaków lub krótsza niż 5, wtedy zostaje wyrzucony błąd.
Dla tego rekordu została przypisana metoda hashująca hasło o nazwie „CalculateMD5Hash”, która jest wykorzystywana przed zapisywaniem hasła do bazy danych.

```
using projekt.src.Exceptions;

namespace projekt.src.Models.Users;

public record AccountType
{

    public AccountType(string value){
        if(string.IsNullOrWhiteSpace(value) || !AvailableRoles.Contains(value.ToLower())) throw new CustomException("Invalid account type.");
        Value = value;
    }

    public string Value { get; }
    private static readonly List<string> AvailableRoles = new List<string> {"admin", "user"};

    public static AccountType Admin(){
        return new AccountType("admin");
    }

    public static AccountType User(){
        return new AccountType("user");
    }
}
```

Rysunek 6 Typ AnnountType

AccountType jest wykorzystywany do przypisania roli użytkownikowi znajdującej się na liście. Dostępnymi opcjami są „admin” oraz „user”. Każda inna wartość zostanie odrzucona.

Kontroler użytkownika

UserController

```
namespace projekt.src.Controllers;

[ApiController]
[Route("[controller]")]
[EnableCors("AllowAnyOrigin")]
public class UsersController : ControllerBase
{
    private readonly ApiDbContext _dbContext;
    private readonly IConfiguration _configuration;
    private readonly UserService _userService;

    public UsersController(
        ApiDbContext dbContext,
        IConfiguration configuration,
        UserService userService
    )
    {
        _dbContext = dbContext;
        _configuration = configuration;
        _userService = userService;
    }

    [HttpPost("register", Name="Create new user")]
    public async Task<IActionResult>Signup([FromBody] RegisterDto userDto)
    {
        var email = new Email(userDto.Email);
        var name = new Name(userDto.Name);
        var lastname = new LastName(userDto.Lastname);
        var password = new Password(userDto.Password);
        var confirmPassword = new Password(userDto.ConfirmPassword);
        var address = string.IsNullOrEmpty(userDto.Address) ? null : new Address(userDto.Address);
        var location = string.IsNullOrEmpty(userDto.Location) ? null : new Address(userDto.Location);
        var postcode = string.IsNullOrEmpty(userDto.PostCode) ? null : new PostCode(userDto.PostCode);
        var phone = string.IsNullOrEmpty(userDto.Phone) ? null : new Phone(userDto.Phone);

        var exists = await _dbContext.Users.AnyAsync(x=>x.Email == email);
        if(exists) throw new CustomException("User with this email already exists.");

        var hashedPassword = new Password(password.CalculateMD5Hash(password.Value));
```

Rysunek 7 Register

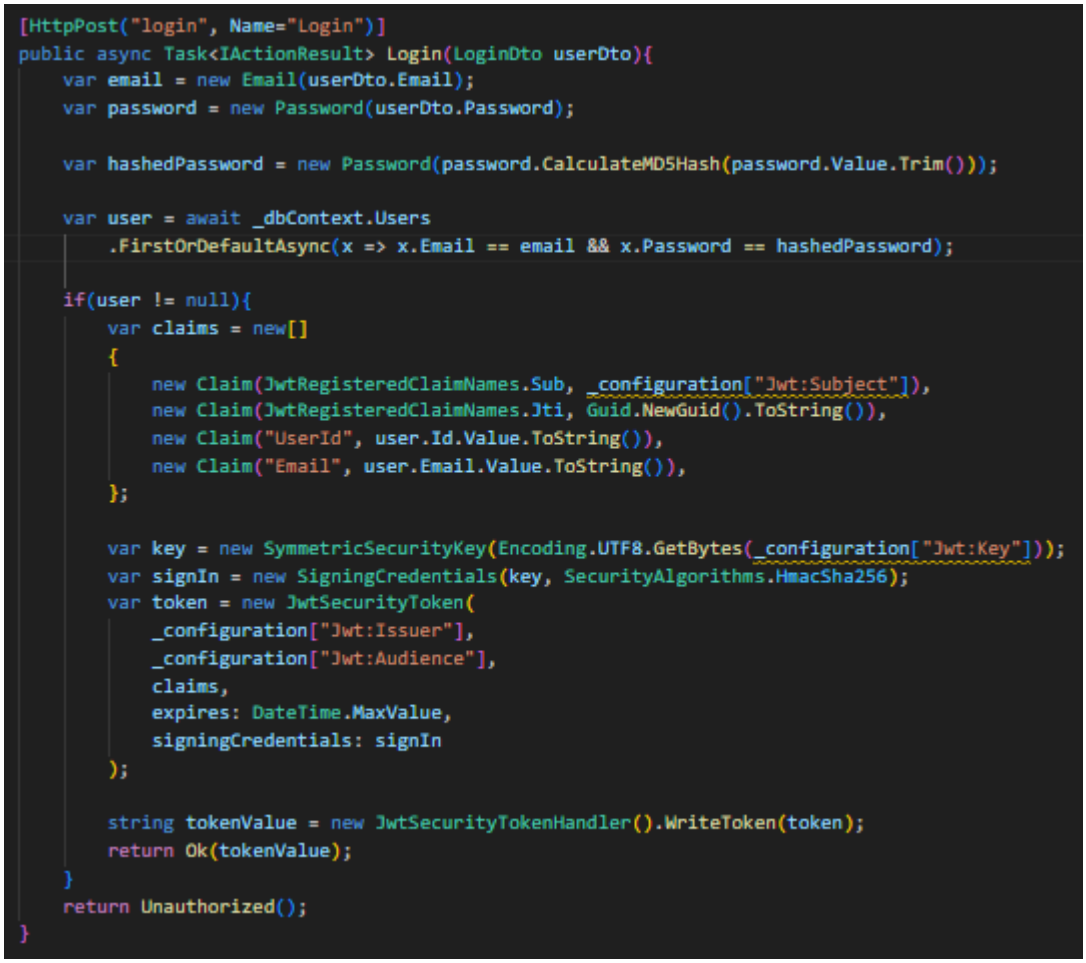
```
    var user = Models.Users.User.NewUser(
        email,
        name,
        lastname,
        hashedPassword,
        address,
        location,
        postcode,
        phone,
        DateTime.UtcNow
    );

    _dbContext.Users.AddAsync(user);
    await _dbContext.SaveChangesAsync();

    return Ok(user);
}
```

Rysunek 8 Register

Na rysunku 7 i 8 przedstawiona jest metoda służąca do rejestracji użytkownika, gdzie wartości zmiennych są odczytywane z przygotowanego DTO rejestracji. Jeśli użytkownik o danym emailu już istnieje, nie jest możliwe stworzenie drugiego konta o tym emailu. Podane hasło jest hashowane, a następnie za pomocą metody w klasie User o nazwie „NewUser” zmienne są przypisywane, a następnie nowy użytkownik jest zapisywany do bazy danych.



```
[HttpPost("login", Name="Login")]
public async Task<IActionResult> Login(LoginDto userDto){
    var email = new Email(userDto.Email);
    var password = new Password(userDto.Password);

    var hashedPassword = new Password(password.CalculateMD5Hash(password.Value.Trim()));

    var user = await _dbContext.Users
        .FirstOrDefaultAsync(x => x.Email == email && x.Password == hashedPassword);

    if(user != null){
        var claims = new[]
        {
            new Claim(JwtRegisteredClaimNames.Sub, _configuration["Jwt:Subject"]),
            new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
            new Claim("UserId", user.Id.Value.ToString()),
            new Claim("Email", user.Email.Value.ToString()),
        };

        var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]));
        var signIn = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
        var token = new JwtSecurityToken(
            _configuration["Jwt:Issuer"],
            _configuration["Jwt:Audience"],
            claims,
            expires: DateTime.MaxValue,
            signingCredentials: signIn
        );

        string tokenValue = new JwtSecurityTokenHandler().WriteToken(token);
        return Ok(tokenValue);
    }
    return Unauthorized();
}
```

Rysunek 9 Login

Rysunek 9 przedstawia kod służący do logowania na konto użytkownika. Podane hasło jest hashowane, a następnie ono oraz email są porównywane z rekordami z bazy danych. Jeśli istnieje użytkowni o takich danych, tworzony jest token zawierający „UserId”, czyli Id użytkownika oraz jego email. Na potrzeby projektu token nie ma daty ważności. Na końcu wartość tokenu jest zwracana. Jeżeli użytkownik o danych wartościach nie istnieje w bazie danych, zwracany jest wyjątek Unauthorized.

```
[HttpGet("me")]
[Authorize]
public async Task<IActionResult> GetMe(){
    var user = await _userService.CurrentUser(User);
    return Ok(user);
}
```

Rysunek 10 Pobranie obecnego użytkownika

Metoda GetMe pobiera i zwraca dane obecnie zalogowanego użytkownika. Wymagana w tym celu jest autoryzacja za pomocą tokenu. Użyte zostaje tutaj repozytorium z użytkownikiem nazwane UserService.

```
[HttpGet("user")]
public async Task<IActionResult> GetById([FromQuery] Guid userId){
    var id = new UserId(userId);
    var user = await _dbContext.Users.FirstOrDefaultAsync(x=>x.Id == id);
    if(user is null) throw new CustomException("User not found.");
    return Ok(user);
}
```

Rysunek 11 Get user by Id

Metoda GetById zwraca użytkownika wyszukiwanego na podstawie jego Id. W przypadku jest braku, metoda zwróci wyjątek „User not found”.

```
[HttpDelete("delete")]
[Authorize]
public async Task<IActionResult> Delete([FromBody] bool confirm){
    if(!confirm) throw new CustomException("Deleting account canceled.");

    var user = await _userService.CurrentUser(User);
    _dbContext.Users.Remove(user);
    await _dbContext.SaveChangesAsync();

    return Ok("Account deleted successfully.");
}
```

Rysunek 12 Delete

Metoda Delete pozwala na usunięcie obecnego użytkownika z bazy danych. Wymagana jest autoryzacja za pomocą tokenu.

```
[HttpPut("changePassword")]
[Authorize]
public async Task<IActionResult> UpdatePassword([FromBody] ChangePasswordDto dto)
{
    var user = await _userService.CurrentUser(User);
    var oldPassword = new Password(dto.OldPassword);
    var oldPasswordHashed = new Password(oldPassword.CalculateMD5Hash(oldPassword.Value.Trim()));

    if(!user.Password.Equals(oldPasswordHashed)) throw new CustomException("Incorrect password.");

    var password = new Password(dto.NewPassword);
    var hashedPassword = new Password(password.CalculateMD5Hash(password.Value));
    if(user.Password.Equals(hashedPassword)) throw new CustomException("New password cannot be the same as old password.");

    user.UpdatePassword(hashedPassword);
    await _dbContext.SaveChangesAsync();

    return Ok("Password changed successfully.");
}
```

Rysunek 13 Change password

Rysunek 13 pokazuje metodę służącą do zmiany hasła. Wymagane jest podanie obecnego hasła, a następnie nowego oraz jego powtórzenie. Jeśli obecne hasło się nie zgadza lub jest takie samo jak stare hasło, zostanie wyrzucony odpowiedni wyjątek. W pliku DTO został zawarty warunek sprawdzający czy nowe hasło oraz powtórzenie nowego hasła są ze sobą zgodne.

```
[HttpPut("changeInformation")]
[Authorize]
public async Task<IActionResult> UpdateInformation([FromBody] ChangeInformationDto dto)
{
    var name = dto.Name is null ? null : new Name(dto.Name);
    var lastname = dto.LastName is null ? null : new LastName(dto.LastName);
    var address = dto.Address is null ? null : new Address(dto.Address);
    var location = dto.Location is null ? null : new Address(dto.Location);
    var postCode = dto.PostCode is null ? null : new PostCode(dto.PostCode);
    var phone = dto.Phone is null ? null : new Phone(dto.Phone);

    var user = await _userService.CurrentUser(User);
    user.UpdateInformation(name, lastname, address, location, postCode, phone);
    await _dbContext.SaveChangesAsync();

    return Ok();
}
```

Rysunek 14 Change information

Ostatnia metoda w kontrolerze użytkownika służy do zmiany informacji takich jak imię, nazwisko, adres, lokalizacja, kod pocztowy czy też numer telefonu. Wykorzystywana jest metoda w klasie User o nazwie „UpdateInformation”.

UserService

```
namespace projekt.src.Controllers;

public class UserService
{
    private readonly ApiDbContext _dbContext;

    public UserService(ApiDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public async Task<User> CurrentUser(ClaimsPrincipal userClaims)
    {
        var userId = userClaims.FindFirst("UserId").Value;
        var user = await _dbContext.Users.FirstOrDefaultAsync(x=>x.Id == userId);

        if(user is null) throw new CustomException("User not found.");
        return user;
    }

    public async Task<bool> CheckIfUSerExistsById(Guid userId)
    {
        var id = new UserId(userId);
        var user = await _dbContext.Users.AnyAsync(x=>x.Id == id);
        return user;
    }

}
```

Rysunek 15 UserService

Rysunek 15 przedstawia metody w klasie UserService, którymi są CurrentUser oraz CheckIfUserExistsById. Pierwsza z nich służy do pobierania obecnie zalogowanego użytkownika na podstawie tokenu użytego w autoryzacji. Użytkownik jest pobierany z bazy danych a następnie jest informacje są zwracane.

Druga metoda służy do sprawdzenia, czy użytkownik istnieje w bazie danych. W tym celu wymagane jest podanie jego Id.

DTO

```
public record ChangePasswordDto
{
    [Required(ErrorMessage = "Password incorrect.")]
    [DataType(DataType.Password)]
    public string OldPassword{get; set;}

    [Required(ErrorMessage = "Password is required.")]
    [Compare("ConfirmNewPassword", ErrorMessage ="Password does not match.")]
    [DataType(DataType.Password)]
    public string NewPassword {get; set;}

    [Required(ErrorMessage = "Confirm password is required.")]
    [DataType(DataType.Password)]
    public string ConfirmNewPassword {get; set;}
}
```

Rysunek 16 Password DTO

W projekcie zastosowano wiele wzorców projektowych jakimi są Data Trasfer Object, czyli DTO. Są one użyte w celu lepszego wyświetlania i selekcjonowania informacji lub też ich pobierania. W tym przykładowym DTO służące do zmiany hasła pokazane są pola do wypełnienia przez użytkownika. Użyta jest tutaj metoda Compare służąca do porównania nowego hasła a powtórzeniem nowego hasła.

ApiDbContext

```
public class ApiDbContext : DbContext
{
    public ApiDbContext(DbContextOptions<ApiDbContext> options)
        : base(options) {}

    public DbSet<User> Users { get; set; }
    public DbSet<Announcement> Announcements { get; set; }
    public DbSet<Reviews> Reviews { get; set; }
    public DbSet<SavedAnnouncements> SavedAnnouncements { get; set; }
    public DbSet<ShoppingCart> ShoppingCarts { get; set; }
    public DbSet<ShoppingCartItem> ShoppingCartItems { get; set; }
    public DbSet<Orders> Orders { get; set; }
    public DbSet<OrderedItems> OrderedItems { get; set; }
```

Rysunek 17 Sety

W strukturze DbContext zawarte są DbSety przedstawiające tabele stworzonych modeli.

OnModelCreating

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<User>(builder =>
    {
        builder.HasKey(x => x.Id);
        builder.Property(x => x.Id).HasConversion(id => id.Value, value => new UserId(value));

        builder.HasIndex(x=>x.Email).IsUnique();
        builder.Property(x=>x.Email).HasConversion(email => email.Value, value => new Email(value));

        builder.Property(x=>x.Name).HasConversion(name => name.Value, value => new Name(value))
            .IsRequired()
            .HasMaxLength(50);

        builder.Property(x=>x.LastName).HasConversion(ln=>ln.Value, value=>new LastName(value))
            .IsRequired()
            .HasMaxLength(200);

        builder.Property(x=>x.Password).HasConversion(p => p.Value, value => new Password(value))
            .IsRequired()
            .HasMaxLength(200);

        builder.Property(x=>x.Address).HasConversion(a => a.Value, value => new Address(value));

        builder.Property(x=>x.Location).HasConversion(Location => Location.Value, value => new Address(value));

        builder.Property(x=>x.PostCode).HasConversion(PostCode => PostCode.Value, value => new PostCode(value));

        builder.Property(x=>x.Phone).HasConversion(Phone =>Phone.Value, value => new Phone(value));

        builder.Property(x=>x.Role).HasConversion(Role => Role.Value, value => new AccountType(value))
            .IsRequired();

        builder.Property(x=>x.CreatedAt).IsRequired();
    });
```

Rysunek 18 OnModelCreating

W metodzie OnModelCreating manualnie zostały przypisane zmienne wykorzystywane w bazie danych, gdzie zadeklarowany zostaje klucz główny, wszystkie wartości modelu oraz ich konwersja. Jest ona wymagana w wielu przypadkach, np. przy zmianie prostego Id z bazy danych na typ UserId używanym w modelu.

Konto administratora

```
var passwordForAdmin = new Password("admin");
var hashedPasswordForAdmin = new Password(passwordForAdmin.CalculateMD5Hash(passwordForAdmin.Value));

var admin = User.NewAdmin(
    email: new Email("admin@example.com"),
    name: new Name("Admin"),
    lastname: new LastName("Admin"),
    password: hashedPasswordForAdmin,
    address: null,
    location: null,
    postcode: null,
    phone: null,
    createdAt: DateTime.UtcNow
);


modelBuilder.Entity<User>().HasData(new
{
    Id = admin.Id,
    Email = admin.Email,
    Name = admin.Name,
    LastName = admin.LastName,
    Password = admin.Password,
    Address = admin.Address,
    Location = admin.Location,
    PostCode = admin.PostCode,
    Phone = admin.Phone,
    Role = admin.Role,
    CreatedAt = admin.CreatedAt
});
```

Rysunek 19 Konto admina

W metodzie OnModelCreating zawarte zostało także tworzenie konta admina za pomocą metody w klasue User jaką jest NewAdmin. Email zostaje ustawione na admin@example.com , a hashowane hasło to admin. Następnie konto zostaje zapisane w bazie danych.

Procedura

```
    [HttpGet]
[Authorize]
public async Task<List<OrderDto>> GetOrdersForMe()
{
    var user = await _userService.CurrentUser(User);

    var ownedAnnouncementsId = await _dbContext.Announcements
                                        .Where(x => x.OwnerId == user.Id)
                                        .Select(x => x.Id)
                                        .ToListAsync();

    var orders = await _dbContext.Set<Orders>()
                            .FromSqlRaw($"SELECT * FROM public.getuserordersfunction('{user.Id.Value}'::UUID)")
                            .Include(x=>x.Items)
                            .ToListAsync();

    var filtered = orders.Select(order => new OrderDto
    {
        Id = order.Id,
        OrderingPerson = order.OrderingPerson,
        OrderedAt = order.OrderedAt,
        DeliveryMethod = order.DeliveryMethod,

        Items = (order.Items ?? new List<OrderedItems>())
                .Where(item => ownedAnnouncementsId.Contains(item.AnnouncementId))
                .ToList()
    }).ToList();

    return filtered;
}
```

Rysunek 20 Wywołanie procedury

```
DROP FUNCTION getuserordersfunction;
CREATE OR REPLACE FUNCTION public.getuserordersfunction(userid uuid)
 RETURNS TABLE("Id" uuid, "OrderingPerson" uuid, "OrderedAt" timestamp with time zone, "DeliveryMethod" text)
 LANGUAGE plpgsql
AS $function$
BEGIN
    CREATE TEMP TABLE TempOwnedAnnouncements AS
    SELECT a."Id"
    FROM "Announcements" a
    WHERE a."OwnerId" = userid;

    RETURN QUERY
    SELECT o."Id", o."OrderingPerson", o."OrderedAt", o."DeliveryMethod"
    FROM "Orders" o
    JOIN "OrderedItems" oi ON o."Id" = oi."OrderId"
    WHERE oi."AnnouncementId" IN (SELECT t."Id" FROM TempOwnedAnnouncements t);

    DROP TABLE TempOwnedAnnouncements;
END;
$function$
```

Rysunek 21 Procedura

Funkcja najpierw tworzy tymczasową tabelę (TempOwnedAnnouncements), w której znajdują się identyfikatory ogłoszeń (Id) należących do użytkownika o podanym userid. Jest to wykonane poprzez zapytanie do tabeli Announcements, filtrując po OwnerId
Funkcja następnie wykonuje zapytanie do tabeli Orders, łącząc ją z tabelą OrderedItems za pomocą JOIN. Dodatkowo, filtruje zamówienia, aby wybrać tylko te, które mają powiązane przedmioty (OrderedItems) z ogłoszeniami

należącymi do użytkownika. To oznacza, że funkcja zwraca zamówienia, które dotyczą ogłoszeń, których właścicielem jest użytkownik.
Po wykonaniu zapytania i zwróceniu wyników funkcja usuwa tymczasową tabelę.

Aplikacja wywołuje tę funkcję, następnie filtruje wyniki, aby powiązać zamówienia z przedmiotami związanymi z ogłoszeniami użytkownika.

Ostatecznie aplikacja zwraca użytkownikowi szczegółowe informacje o zamówieniach z jego ogłoszeń od innych użytkowników w postaci obiektów OrderDto.