Programowanie Aplikacji Webowych

Dokumentacja projektu

Bartłomiej Mazurkiewicz,

173670, 3EF-DI/AA, L02

Spis treści

Backend

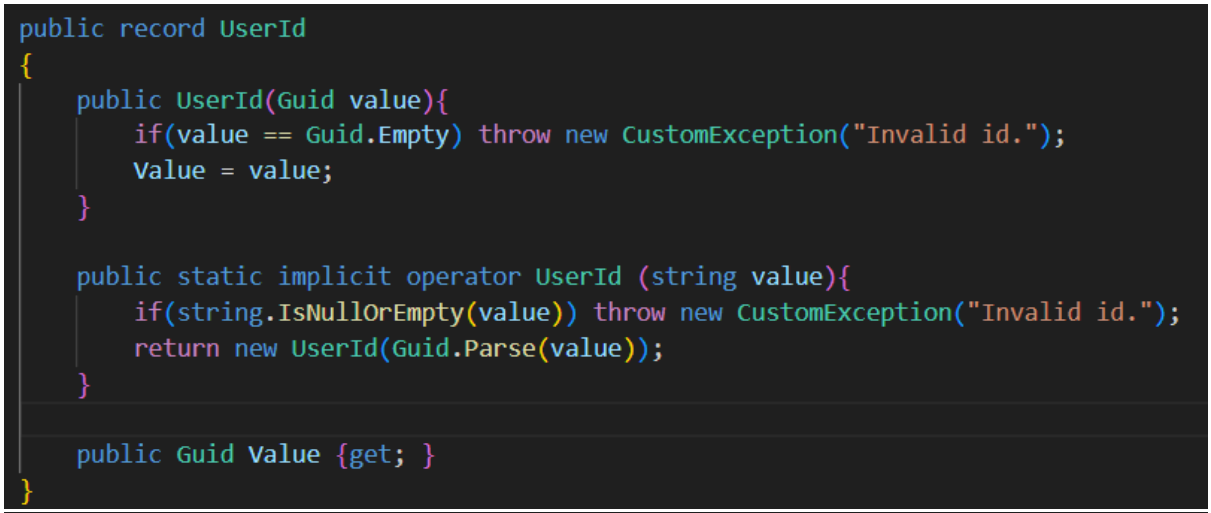
User model

```
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; }

    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);
    }
}
```

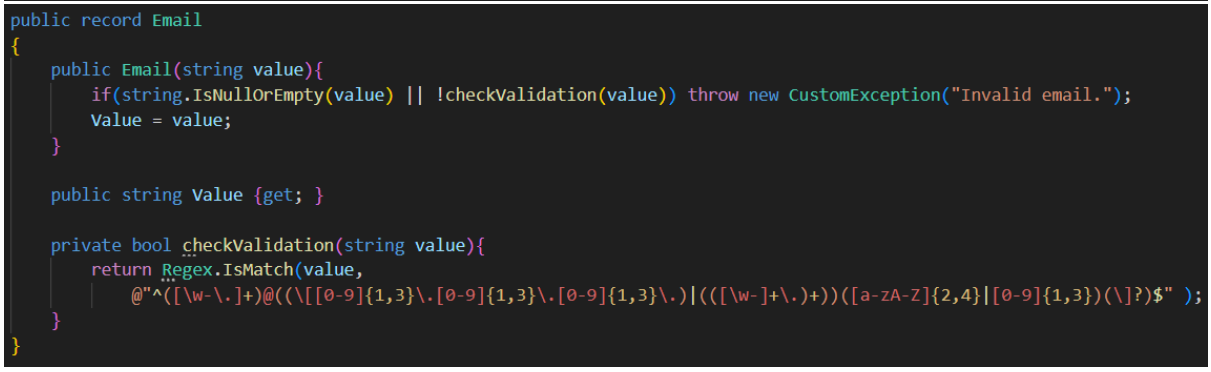
Model użytkownika zawiera takie wartości jak Id, Email, Name, LastName, Password, Role, CreatedAt, które są wymagane przy tworzeniu obiektu. Występują także wartości opcjonalne, takie jak Address, Locaion, PostCode oraz Phone.

Każda z tych wartości ma swój własny zmiennej, przykładowo zmienna Id jest typu UserId, Email typu Email, Name typu Name, LastName typu LastName, Password typu Password, Address oraz Location typu Address, Phone typu Phone, Role typu AccountType.

Każdy typ umieszczony jest w osobnym pliku, gdzie występuj walidacja lub sprawdzenie, czy wartość przypisywana nie jest pusta.



```
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; }
}
```



```
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-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$" );
    }
}
```

```
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");
    }
}
```

W przypadku typu AccountType przypisanie nowej roli użytkownikowi następuje w prosty sposób, jakim jest podanie nazwy rekordu a następnie nazwy roli, przykładowo AnnountType.User(); Rolę można dodać jedynie taką, jaka znajduje się na liście. Obecnie znajdują się na niej dwie pozycje: user oraz admin.

W klasie User występuje konstruktor służący stworzeniu nowego obiektu użytkownika przypisując odpowiednie zmienne.

Zaimplementowane zostały także dwie metody służące do tworzenia użytkownika z odpowiednią rolą. Metoda NewUser() używana jest w kontrolerze do tworzenia nowego użytkownika ze zwykłymi uprawnieniami, natomiast metoda NewAdmin() jest wykorzystywana przy tworzeniu projektu w celu automatycznego stworzenia konta Administratora.

```
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();
    }
}
```

Rekord Password zawiera metodę CalculateMD5Hash służącą do hashowania haseł w celu bezpieczeństwa.

Announcement model

```
public class Announcement
{
    public Announcement(){}
    private Announcement(Guid id, UserId ownerId, Item item, DateTime createdAt)
    {
        Id = id;
        OwnerId = ownerId;
        Item = item;
        CreatedAt = createdAt;
    }

    public Guid Id { get; private set;}
    public UserId OwnerId{ get; private set;}
    public Item Item{get; private set;}
    public DateTime CreatedAt { get; private set;}

    public static Announcement New(UserId ownerId, Item item)
    {
        return new Announcement(Guid.NewGuid(), ownerId, item, DateTime.UtcNow);
    }
}
```

Model ogłoszenia zawiera wartości takie jak Id, OwnerId typu UserId, obiekt Item, CreatedAt. W celu stworzenia obiektu wykorzystywany jest konstruktor przypisujący odpowiednie pola. Klasa zawiera metodę New() wykorzystywaną do tworzenia nowego ogłoszenia.

Item model

```
public class Item
{
    private Item(){}

    private Item(string title, string? description, ItemAmount amount, Cost cost)
    {
        Title = title;
        Description = description;
        Amount = amount;
        Cost = cost;
    }

    public string Title { get; private set; }
    public string? Description { get; private set; }
    public ItemAmount Amount { get; private set; }
    public Cost Cost { get; private set; }
    public DateTime? UpdatedAt { get; private set; }

    public static Item NewItem(
        string title, string? description, ItemAmount amount, Cost cost)
    {
        return new Item(title, description, amount, cost);
    }

    public void UpdateItem(
        string? title, string? description, ItemAmount? amount, Cost? cost
    )
    {
        Title = title;
        Description = description;
        Amount = amount;
        Cost = cost;
        UpdatedAt = DateTime.UtcNow;
    }

}
```

Obiekt Item jest wykorzystywany w klasie Announcement. Zwawiera pola Title, Description, Amount typu ItemAmount, Cost typu Cost oraz UpdatedAt. Zawarta jest tutaj metoda NewItem() tworząca nowy przedmiot oraz metoda UpdatedItem wykorzystywana do aktualizowania przedmiotu.

```
public record ItemAmount
{
    public ItemAmount(int value)
    {
        if(value < 0) throw new CustomException("Invalid amount.");
        Value = value;
    }
    public int Value { get; }

}
```

```
public record Cost
{
    public Cost(decimal value){
        if(value < 0) throw new CustomException("Invalid cost.");
        Value = value;
    }

    public static Cost operator +(Cost a, Cost b)
    {
        if(a is null || b is null ) throw new CustomException("Invalid cost");
        return new Cost(a.Value + b.Value);
    }

    public static Cost operator*(Cost a, int b)
    {
        return new Cost(a.Value*b);
    }

    public decimal Value {get; }
}
```

Zarówno w rekordzie ItemAmount jak i Cost jest sprawdzane, czy wartości nie są ustawione poniżej 0. W rekordzie Cost zaimplementowane zostały operatory dodawania i mnożenia, aby można było dodawać lub mnożyc dwa obiekty typu Cost.

ShoppingCart model

```
public class ShoppingCart
{
    private ShoppingCart(Guid id, UserId ownerId)
    {
        Id = id;
        OwnerId = ownerId;
        UpdatedAt = DateTime.UtcNow;
    }

    private ShoppingCart(Guid id, UserId ownerId, List<ShoppingCartItem> items, DateTime updatedAt)
    {
        Id = id;
        OwnerId = ownerId;
        Items = items;
        UpdatedAt = updatedAt;
    }

    public Guid Id { get; private set; }
    public UserId OwnerId { get; private set; }
    public List<ShoppingCartItem> Items {get; private set; } = new List<ShoppingCartItem>();
    public DateTime UpdatedAt { get; private set; }

    public static ShoppingCart New(UserId ownerId)
    {
        return new ShoppingCart(Guid.NewGuid(), ownerId);
    }

    public ShoppingCartItem AddItem(Guid announcementId, Quantity quantityVal)
    {

        var quantity = new Quantity(quantityVal.Value);
        var cartItem = ShoppingCartItem.NewItem(Id, announcementId, quantity);
        UpdatedAt = DateTime.UtcNow;
        Items.Add(cartItem);
        return cartItem;
    }
```

```
    public void UpdateItem(Guid announcementId, Quantity quantityVal)
    {
        var quantity = new Quantity(quantityVal.Value);
        var cartItem = Items.FindIndex(x=>x.AnnouncementId == announcementId);

        UpdatedAt = DateTime.UtcNow;
        Items[cartItem].Update(quantity);
    }

    public void ClearCart()
    {
        UpdatedAt = DateTime.UtcNow;
        Items.Clear();
    }

    public void RemoveOneItem(Announcement item)
    {
        UpdatedAt = DateTime.UtcNow;
        var index = Items.FindIndex(x=>x.AnnouncementId == item.Id);
        Items.RemoveAt(index);
    }
}
```

Model koszyka zawiera pola Idm OwnerId typu UserId, listę Items typu ShoppingCartITem oraz UpdatedAt.

Zaimplementowana metoda New() służy do stworzenia nowego koszyka dla użytkownika.

AddItem to metoda dodająca do koszyka nowy przedmiot.

UpdateItem służy do aktualizowania przedmiotów w koszyku.

ClearCart służy do wyczyszczenia koszyka ze wszystkich przedmiotów.

RemoveOneItem usuwa wybrany przedmiot z koszyka.

```
public class ShoppingCartItem
{
    public ShoppingCartItem(){}
    private ShoppingCartItem(Guid id, Guid shoppingCartId, Guid announcementId, Quantity quantity)
    {
        Id = id;
        ShoppingCartId = shoppingCartId;
        AnnouncementId = announcementId;
        Quantity = quantity;
    }

    public Guid Id { get; private set; }
    public Guid ShoppingCartId { get; private set; }
    public Guid AnnouncementId { get; private set; }
    public Quantity Quantity { get; private set; }

    public static ShoppingCartItem NewItem(Guid shoppingCartId, Guid announcementId, Quantity quantity)
    {
        return new ShoppingCartItem(Guid.NewGuid(), shoppingCartId, announcementId, quantity);
    }

    public void Update(Quantity quantity)
    {
        Quantity = quantity;
    }
}
```

ShoppingCartItem to klasa zawierająca Id, ShoppingCartId, AnnouncementId oraz Quantity typu Quantity. Znajduje się tutaj metoda NewItem() do tworzenia nowego przedmiotu dla koszyka.

Metoda update aktualizuje ilość wybranego przedmiotu.

```
public record Quantity
{
    public Quantity(int value)
    {
        if(value < 0) throw new CustomException("Invalid quantity of item.");
        Value = value;
    }

    public static Quantity operator +(Quantity a, Quantity b){
        if(a == null || b == null) throw new CustomException("Invalid quantity.");
        return new Quantity(a.Value + b.Value);
    }

    public int Value { get; }

    public int getValue() => Value;
}
```

Quantity to rekord zawierający operator dodawania oraz metodę do zwracania wartości nazwaną getValue().

User Controller

Register

```
[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));

    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);
}
```

Customowa metoda Register służy do rejestracji nowego użytkownika. Wykorzystywane jest DTO w celu pobrania wartości od użytkownika i przypisania ich do odpowiednich zmiennych. Hasło przed stworzeniem nowego obiektu jest hashowane, a dopiero następnie przesyłane do metody torzącej użytkownika.

Login

```
[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();
}
```

Własnoręcznie zaimplementowana metoda służąca do logowania pobiera użytkownika z bazy danych na podstawie emailu oraz porównaniu zahashowanych haseł. W przypadku braku rekordu w bazie danych zwracany jest kod Unauthorized.

Jeśli obiekt istnieje w bazie, tworzony jest token posiadający wartość UserId oraz Email. Na potrzeby projektu token nigdy nie wygasa. Po udanym logowaniu token jest zwracany i gotowy do pobrania.

Metody GetMe, GetById, Delete

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

[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);
}

[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.");
}
```

Metody GetMe() oraz Delete() posługują się repozytorium UserService zawierające informacje na temat zalogowanego użytkownika. GetMe() to metoda służąca do wyświetlenia informacji o obecnie zalogowanym użytkowniku. Metoda Delete() usuwa obecnie zalogowaneg użytkownika. W obu przypadkach wymagana jest wcześniejsza autoryzacja za pomocą tokenu.

Metoda GetById() pozwala na pobranie informacji użytkownika, którego Id zostanie podane.

UserService

```
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;
    }

}
```

UserService posiada metodę CurrentUser, która zwraca obiekt użytkownia pobranego z bazy danych na podstawie użytego tokenu. Odczytywany jest z niego Id, a następnie pobiera obiekt User z bazy danych i zwraca go.

Metoda CheckIfUSerExistsById() sprawdza, czy dany użytkownik istnieje na podstawie podanego Id.

ShoppingCart Controller

CreateCart, AddItem, GetCart

```
[NonAction]
public async Task <ShoppingCart> CreateCart()
{
    var user = await _userService.CurrentUser(User);
    var cartExists = await GetCart();

    if(cartExists is null){
        var cart = ShoppingCart.New(user.Id);

        _dbContext.ShoppingCarts.Add(cart);
        await _dbContext.SaveChangesAsync();

        return cart;
    }

    return cartExists;
}

[HttpPost]
[Authorize]
public async Task<IActionResult> AddItem(
    [FromQuery] Guid AnnouncementId,
    [FromBody] AddItemDto dto
)
{
    var announcement = await _dbContext.Announcements.FirstOrDefaultAsync(x => x.Id == AnnouncementId);
    if (announcement is null)
        throw new CustomException("Announcement not found.");

    var user = await _userService.CurrentUser(User);
    var cart = await CreateCart();
    if (cart is null)
        throw new CustomException("Cart not found.");

    var quantity = new Quantity(dto.Quantity);

    if(dto.Quantity <= 0) throw new CustomException("Invalid amount.");

    var totalAvailableQuantity = announcement.Item.Amount.Value;
    if (quantity.Value > totalAvailableQuantity)
        throw new CustomException("The selected quantity exceeds the available amount of the item.");


     var existingItem = cart.Items.FirstOrDefault(x => x.AnnouncementId == announcement.Id);

    if (existingItem != null)
    {
        existingItem.Update(quantity);
        _dbContext.ShoppingCartItems.Update(existingItem);
    }
    else
    {
        var cartItem = ShoppingCartItem.NewItem(cart.Id, AnnouncementId, quantity);
        _dbContext.ShoppingCartItems.Add(cartItem);
    }

    await _dbContext.SaveChangesAsync();
    return Ok(cart);
}
```

```
[HttpGet]
[Authorize]
public async Task <ShoppingCart> GetCart()
{
    var user = await _userService.CurrentUser(User);
    var cart = await _dbContext.ShoppingCarts
                        .Include(x=>x.Items)
                        .FirstOrDefaultAsync(x=>x.OwnerId == user.Id);

    return cart;
}
```

Kontroller posiada metodę AddItem, gdzie dodaje przedmiot do koszyka użytkownika. Jeśli koszyk nie istnieje, wykorzystuje metodę CreateCart do stworzenia go, natomiast jeśli koszyk istnieje, to wykorzystuję tę motodę do zwrócenia obiektu cart wywołując metodę GetCart(). Jeśli przy dodwaniu przedmiotu dany przedmiot już istnieje w koszyku, wykorzystywana jest metoda do aktualizacji ilości przedmiotu za pomocą metody Update().

UpdateItem

```
[HttpPut]
[Authorize]
public async Task <IActionResult> UpdateItem(
    [FromQuery] Guid AnnouncementId,
    [FromBody] AddItemDto dto)
{
    var user = await _userService.CurrentUser(User);
    var cart = await GetCart();

    var existingItem = cart.Items.FirstOrDefault(x=>x.AnnouncementId == AnnouncementId);

    if(existingItem is null) throw new CustomException("Item not found.");

    var quantity = new Quantity(dto.Quantity);

    if(dto.Quantity <= 0) throw new CustomException("Invalid amount.");

    var announcement = await _dbContext.Announcements.FirstOrDefaultAsync(x => x.Id == AnnouncementId);
    if (announcement is null)
        throw new CustomException("Announcement not found.");

    var totalAvailableQuantity = announcement.Item.Amount.Value;
    if (quantity.Value > totalAvailableQuantity)
        throw new CustomException("The selected quantity exceeds the available amount of the item.");

    cart.UpdateItem(AnnouncementId, quantity);

    await _dbContext.SaveChangesAsync();

    return Ok(cart);
}
```

Metoda UpdateItem() służy do aktualizacji ilości przedmiotu w koszyku. Sprawdzany jest tutaj warunek, czy podana wartość nie przekracza ogólnej ilości dostępnego towaru.

ClearCart

```
[HttpDelete("clear")]
[Authorize]
public async Task<IActionResult> ClearCart()
{
    var cart = await GetCart();

    if(cart is null) throw new CustomException("Cart not found.");

    _dbContext.ShoppingCartItems.RemoveRange(cart.Items);
    cart.ClearCart();
    await _dbContext.SaveChangesAsync();

    return Ok();

}
```

Metoda ClearCart słuzy do wyczyszczenia zawartości koszyka. Użyta zostaje metoda ClearCart znajdująca się w modelu koszyka.

Announcement Controller

Create announcement

```
[HttpPost]
[Authorize]
public async Task<IActionResult> CreateAnnouncement([FromBody] CreateAnnouncementDto dto)
{
    var user = await _userService.CurrentUser(User);

    var itemAmount = new ItemAmount(dto.Amount);
    var cost = new Cost(dto.Cost);

    if(dto.Title is null || dto.Amount <= 0 || dto.Cost is <= 0) throw new CustomException("You need to enter valid options.");

    var item = Item.NewItem(
        dto.Title,
        dto.Description,
        itemAmount,
        cost
    );

    var announcement = Announcement.New(
        user.Id,
        item
    );

    _dbContext.Announcements.AddAsync(announcement);
    await _dbContext.SaveChangesAsync();

    return Ok(announcement);
}
```

Create Announcement służy do stworzenia nowego ogłoszenia, gdzie użytkownik podaje tytuł, opis, koszt oraz ilość przedmiotów.

Tworzony jest tutaj obiekt przedmiotu, a następnie ogłoszenie wraz z nowym obiektem item.

GetMyAnnouncements, GetAnnouncements

```
[HttpGet("my")]
[Authorize]
public async Task<IActionResult> GetMyAnnouncements()
{
    var user = await _userService.CurrentUser(User);
    var announcements = await _dbContext.Announcements
                                        .Where(x=>x.OwnerId == user.Id)
                                        .ToListAsync();

    return Ok(announcements);
}

[HttpGet]
public async Task<IActionResult> GetAnnouncements(
    [FromQuery] Guid? userId,
    [FromQuery] Guid? announcementId
)
{
    var announcements = await _dbContext.Announcements.ToListAsync();

    if(userId is not null){
        var exists = await _userService.CheckIfUSerExistsById(userId.Value);
        if(!exists) throw new CustomException("User does not exist.");

        var id = new UserId(userId.Value);
        announcements = await _dbContext.Announcements
                                        .Where(x=>x.OwnerId == id)
                                        .ToListAsync();
    }

    if(announcementId is not null){
        announcements = await _dbContext.Announcements
                                        .Where(x=>x.Id == announcementId)
                                        .ToListAsync();

        if(announcements is null) throw new CustomException("Announcement not found.");
    }

    return Ok(announcements);
}
```

Pierwsza metoda pobiera dla zalogowanego użytkownika jego własne stworzone ogłoszenia i wyświetla informacje o nich.

Druga metoda sprawdza, czy zostało podane id userId lub announcementId. Jeśli userId nie jest puste, to wyświetla ogłoszenia dla podanego użytkownika. Jeśli announcementI nie jest puste, to wyświetla ogłoszenie na podstawie podanego Id, natomiast jeśli obie wartości są puste, to wyświetlone zostają wszystkie istniejące ogłoszenia.

Update, Delete

```
[HttpPut]
[Authorize]
public async Task <IActionResult> Update(
    [FromQuery] Guid itemId,
    [FromBody] UpdateAnnouncementDto dto
)
{
    var announcement = await _dbContext.Announcements.FirstOrDefaultAsync(x=>x.Id==itemId);
    if(announcement is null) throw new CustomException("Announcement not found.");

    var user = await _userService.CurrentUser(User);
    if(announcement.OwnerId != user.Id && user.Role != AccountType.Admin()) throw new CustomException("You dont have persmission to update this announcement.");

    var itemAmount = new ItemAmount(dto.Amount.Value);
    var cost = new Cost(dto.Cost.Value);

    if(dto.Title is null || dto.Amount <= 0 || dto.Cost is <= 0) throw new CustomException("You need to enter valid options.");

    announcement.Item.UpdateItem(
        dto.Title,
        dto.Description,
        itemAmount,
        cost
    );

    await _dbContext.SaveChangesAsync();

    return Ok(announcement);
}

[HttpDelete]
[Authorize]
public async Task<IActionResult> Delete([FromQuery] Guid announcementId)
{
    var announcement = await _dbContext.Announcements.FirstOrDefaultAsync(x=>x.Id == announcementId);
    if(announcement is null) throw new CustomException("Announcement not found.");

    var user = await _userService.CurrentUser(User);
    if(announcement.OwnerId != user.Id) throw new CustomException("You dont have permission to delete this announcement.");

    _dbContext.Remove(announcement);
    await _dbContext.SaveChangesAsync();

    return NoContent();
}
```

Metody służą do aktualizacji ogłoszenia oraz jest usuwania. Aktualizacji może dokonać jedynie użytkownik, który stworzyć ogłoszenie lub administrator.

DbContext

```
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();
    });
```

OnModelCreating zawiera model tworzenia obiektów. W tym przypadku model User wymaga konwersji własnych typów na typy proste. Wykorzystywana jest w tym celu metoda HasConversion, gdzie przykładowo wartość typowego guid id zamieniona zstaje na UserId.

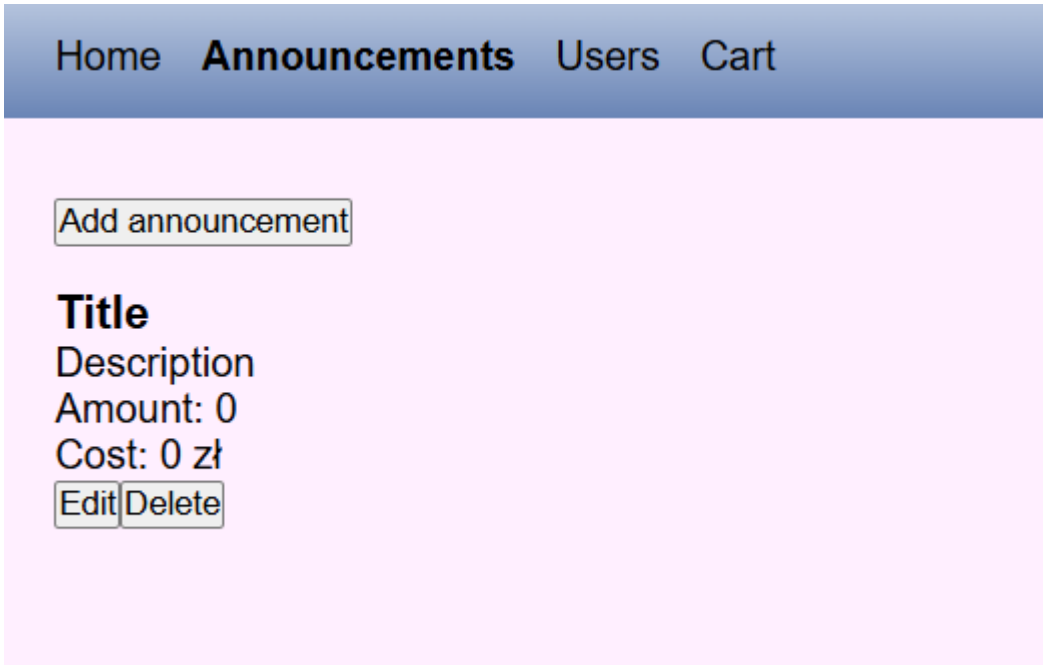
```
var passwordForAdmin = new Password("admin");
var hashedPasswordForAdmin = new Password(passwordForAdmin.CalculateMD5Hash(passwordForAdmin.Value));

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
});
```

Występuje tutaj także implementacja tworzenia konta administratora przy tworzeniu projektu z emailem admin@example.com oraz zahashowanym hasłem admin.

Frontend

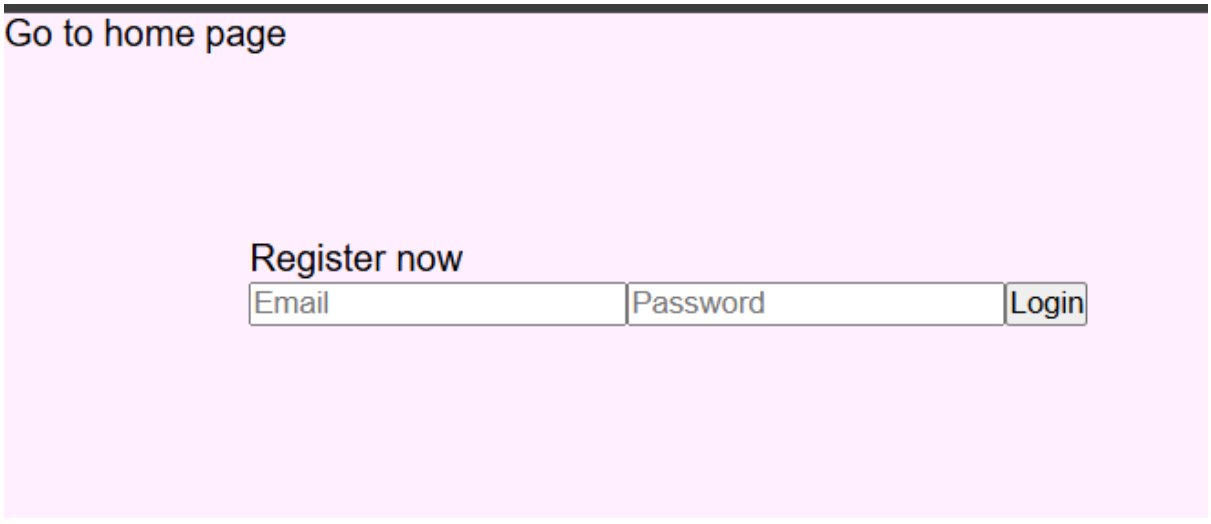
Po wejściu na stronę pojawia się komunikat „Log in to see your profile”.

Z tego miejsca można wyświetlić ogłoszenia w zakładce Announcements.



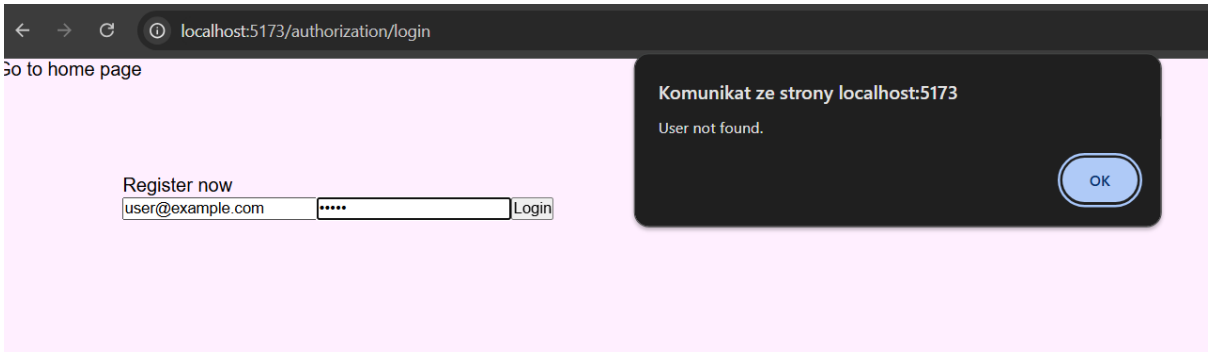
Możliwe jest zarejestrowanie nowego konta w formularzu rejestracji

Można także przejść do panelu logowania klikając „Login now”



Klikając „register now” wrócimy do panelu rejestracji.

Po nieudanym logowaniu wyświetlony zostaje komunikat o błędzie.



Po udanym logowaniu wyświetlona zostają informacje o koncie na stronie głównej, a w nawigacji wyświetlane jest imię oraz nazwisko, a także opcja wylogowania.

Użytkownik został stworzony poprzez dodanie kontekstu UserContext, dlatego wszystkie metody są zawarte w jednym miejscu i mogą zostaćużyte poprzez destrukturyzację.

```jsx
import { createContext, useEffect, useState } from "react";

export const UserContext = createContext();

export const UserProvider = ({children}) => {
    const [user, setUser] = useState(null);

    useEffect(()=>{
        const token = localStorage.getItem('token')
        if(token){
            fetchUser(token)
        }
    }, [])

    const fetchUser = async (token) => {
        try{
            const response = await fetch('http://localhost:5050/Users/me', {
                method: 'GET',
                headers:{
                  'Content-Type': 'application/json',
                   Authorization: `Bearer ${token}`
                }
            })

            if(!response.ok){
                const errorData = response.json();
                console.log(errorData);
                alert("Error in response.")
            }

            const userData = await response.json()
            console.log(userData)
            setUser(userData)
        }catch(error){
            console.log(error)
            alert("Error in catch.")
        }
    }
```

```jsx
const login = async (token) => {
    localStorage.setItem("token", token)
    fetchUser(token)
}

const logout = () => {
    setUser(null);
    localStorage.removeItem('token');
}
```

```
    const register = async (registerData) => {
        console.log(JSON.stringify(registerData))
        try
        {
          const response = await fetch('http://localhost:5050/Users/register', {
            method: 'POST',
            headers:{
              'Content-Type': 'application/json',
            },
            body: JSON.stringify(
                registerData
            )
          });

          if(!response.ok){
            const errorData = await response.json();
            console.log(errorData)
            alert("Error while registering.");
            return;
          }

          alert("Account created.")
        }
        catch(error)
        {
          alert("Error")
          console.log(error)
        }
    }

    return(
        <UserContext.Provider value={{user, login, logout, register}}>
            {children}
        </UserContext.Provider>
    );
}
```

Metody są zatem łatwo dostępne i proste w konfiguracji.