Przewodnik po .NET

Kacper Renkel

August 11, 2024

Contents

1 Wprowadzenie

.NET to platforma programistyczna stworzona przez Microsoft, która pozwala na tworzenie aplikacji na różne platformy. W tym dokumencie omówimy najważniejsze komponenty w architekturze aplikacji .NET, takie jak kontrolery, serwisy, repozytoria, middleware oraz bezpieczeństwo za pomoca JWT.

2 Kontrolery

Kontrolery w .NET odpowiadaja za obsługe żadań HTTP i zwracanie odpowiednich odpowiedzi. Zazwyczaj sa to klasy, które dziedzicza po `ControllerBase` lub `Controller`.

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    private readonly IUserService _userService;

    public UsersController(IUserService userService)
    {
        _userService = userService;
    }

    [HttpGet("{id}")]
    public IActionResult GetUser(int id)
    {
        var user = _userService.GetUserById(id);
        if (user == null)
        {
            return NotFound();
        }
        return Ok(user);
    }
}
```

Listing 1: Przykład kontrolera w .NET

3 Serwisy

Serwisy w .NET służa do enkapsulacji logiki biznesowej. Sa one rejestrowane w kontenerze Dependency Injection (DI) i wstrzykiwane do kontrolerów.

```
public interface IUserService
{
    User GetUserById(int id);
}
```

```
public class UserService : IUserService
{
    private readonly IUserRepository _userRepository;

    public UserService(IUserRepository userRepository)
    {
        _userRepository = userRepository;
    }

    public User GetUserById(int id)
    {
        return _userRepository.FindById(id);
    }
}
```

Listing 2: Przykład serwisu w .NET

4 Repozytoria

Repozytoria w .NET sa odpowiedzialne za komunikacje z baza danych. Służa do abstrakcji operacji na danych, co ułatwia testowanie oraz utrzymanie aplikacji.

```
public interface IUserRepository
{
    User FindById(int id);
}

public class UserRepository : IUserRepository
{
    private readonly ApplicationDbContext _context;

    public UserRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public User FindById(int id)
    {
        return _context.Users.Find(id);
    }
}
```

Listing 3: Przykład repozytorium w .NET

5 Łaczenie z baza danych

W .NET najcześciej używa sie Entity Framework (EF) do komunikacji z baza danych. Konfiguracja połaczenia z baza danych odbywa sie w `Startup.cs` lub `Program.cs`.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services
       )
    {
        services.AddDbContext<ApplicationDbContext>(options
           =>
            options.UseSqlServer(Configuration.
               GetConnectionString("DefaultConnection")));
    }
}
```

Listing 4: Konfiguracja Entity Framework

6 JWT (JSON Web Token)

JWT jest standardem bezpiecznego przesyłania informacji pomiedzy stronami jako obiekt JSON. W .NET JWT sa czesto używane do autoryzacji użytkowników.

6.1 Konfiguracja JWT

Aby skonfigurować JWT w .NET, należy dodać odpowiednie usługi w `Startup.cs` lub `Program.cs`.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme = JwtBearerDefaults
           .AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.
           AuthenticationScheme;
    })
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
           TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Audience"],
```

```
            IssuerSigningKey = new SymmetricSecurityKey(
                Encoding.UTF8.GetBytes(Configuration["Jwt:Key
                "]))
        };
    });

    services.AddControllers();
}
```

Listing 5: Konfiguracja JWT

6.2 Generowanie JWT

Aby wygenerować JWT, należy stworzyć metode, która stworzy token na podstawie danych użytkownika.

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

    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, user.Username)
            ,
        new Claim(JwtRegisteredClaimNames.Email, user.Email),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid()
            .ToString())
    };

    var token = new JwtSecurityToken(
        issuer: _configuration["Jwt:Issuer"],
        audience: _configuration["Jwt:Audience"],
        claims: claims,
        expires: DateTime.Now.AddMinutes(30),
        signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Listing 6: Przykład generowania JWT

7 Hashowanie haseł

W .NET do hashowania haseł można użyć wbudowanego narzedzia `PasswordHasher`, które jest cześcia biblioteki `Microsoft.AspNetCore.Identity`.

```
public class UserService : IUserService
{
    private readonly IUserRepository _userRepository;
    private readonly IPasswordHasher<User> _passwordHasher;

    public UserService(IUserRepository userRepository,
       IPasswordHasher<User> passwordHasher)
    {
        _userRepository = userRepository;
        _passwordHasher = passwordHasher;
    }

    public void RegisterUser(User user, string password)
    {
        user.PasswordHash = _passwordHasher.HashPassword(user
           , password);
        _userRepository.Add(user);
    }
}
```

Listing 7: Przykład hashowania hasła

8 Middlewary

Middlewary w .NET to komponenty, które sa używane do przetwarzania żadań HTTP na różnych etapach ich cyklu życia. Można je wykorzystać np. do logowania, autoryzacji lub obsługi błedów.

```
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;

    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"Request for {context.Request.Path
           } received at {DateTime.Now}");
        await _next(context);
        Console.WriteLine($"Response for {context.Request.
           Path} sent at {DateTime.Now}");
    }
}
```

```
// W Startup.cs
public void Configure(IApplicationBuilder app,
   IHostingEnvironment env)
{
    app.UseMiddleware<LoggingMiddleware>();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Listing 8: Przykład tworzenia middleware

9 Guardy

Guardy to mechanizmy w .NET, które można używać do ochrony określonych zasobów przed nieautoryzowanym dostepem. Można je implementować za pomoca polityk autoryzacyjnych.

```
public class AdminGuard : IAuthorizationRequirement
{
}

public class AdminGuardHandler : AuthorizationHandler<
   AdminGuard>
{
    protected override Task HandleRequirementAsync(
       AuthorizationHandlerContext context, AdminGuard
       requirement)
    {
        if (context.User.IsInRole("Admin"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}

// W Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy("AdminOnly", policy => policy.
           Requirements.Add(new AdminGuard()));
    });
```

```
    services.AddSingleton<IAuthorizationHandler,
       AdminGuardHandler>();
}
```

Listing 9: Przykład implementacji guardu

10 Inne ważne aspekty

10.1 Konfiguracja Dependency Injection (DI)

Dependency Injection (DI) to mechanizm pozwalajacy na wstrzykiwanie zależności do komponentów takich jak serwisy, kontrolery czy repozytoria.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IUserService, UserService>();
    services.AddScoped<IUserRepository, UserRepository>();
    services.AddSingleton<ILogger, Logger>();
}
```

Listing 10: Przykład konfiguracji DI w Startup.cs

10.2 Obsługa wyjatków globalnych

W .NET można skonfigurować globalna obsługe wyjatków za pomoca middleware.

```
public class ExceptionMiddleware
{
    private readonly RequestDelegate _next;

    public ExceptionMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            await HandleExceptionAsync(context, ex);
        }
    }

    private Task HandleExceptionAsync(HttpContext context,
       Exception exception)
```

```
    {
        context.Response.ContentType = "application/json";
        context.Response.StatusCode = (int)HttpStatusCode.
           InternalServerError;

        var result = JsonConvert.SerializeObject(new { error
           = exception.Message });
        return context.Response.WriteAsync(result);
    }
}

// W Startup.cs
public void Configure(IApplicationBuilder app)
{
    app.UseMiddleware<ExceptionMiddleware>();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Listing 11: Przykład globalnej obsługi wyjatków

10.3 Swagger - dokumentacja API

Swagger to narzedzie do automatycznego generowania dokumentacji API. W .NET można je łatwo skonfigurować.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API
           ", Version = "v1" });
    });
}

public void Configure(IApplicationBuilder app)
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API
            V1");
    });
}
```

Listing 12: Konfiguracja Swagger w Startup.cs

11 Podsumowanie

.NET jest wszechstronna platforma, która pozwala na tworzenie rozbudowanych aplikacji webowych z użyciem nowoczesnych wzorców projektowych. Dobre zrozumienie takich komponentów jak serwisy, repozytoria, middleware czy mechanizmy autoryzacji i autentykacji pozwala na budowanie skalowalnych i bezpiecznych aplikacji.