Dezvoltarea Aplicațiilor Web utilizând ASP.NET Core MVC
Curs 11

Cuprins

REST API și Implementarea unei Aplicații CRUD cu ASP.NET Core și Identity

Crearea Proiectului în Visual Studio

Pentru a crea o aplicație REST API cu ASP.NET Core și Identity, se procedează astfel:

- Se deschide Visual Studio și se selectează ***Create a new project***
- Se alege ***ASP.NET Core Web API*** și se apasă ***Next***
- Se introduce un nume pentru proiect (ex: Laborator11RESTAPI) și locația unde va fi salvat
- Se selectează .NET 9 ca framework
- Se activează opțiunile:
    - **Enable OpenAPI support** (pentru documentarea API-ului cu Swagger)

      Swagger – oferă o interfață ușor de utilizat care permite dezvoltatorilor și utilizatorilor să interacționeze cu un API, fără a fi nevoie să scrie cereri manual în Postman sau alte unelte. Swagger reprezintă un set de instrumente și specificații utilizate pentru proiectarea, documentarea și testarea API-urilor. Este unul dintre cele mai populare standarde pentru documentarea API-urilor RESTful, oferind o interfață interactivă care ajută dezvoltatorii să înțeleagă și să testeze API-ul.

    - **Configure for HTTPS**

- **Use controllers (este deja bifata)** – această opțiune rămâne bifată deoarece configurează proiectul să folosească **Controllers**, acestea fiind componentele principale pentru gestionarea logicii API-ului.

➢ La final se apasă **Create**

Așadar, proiectul va include automat o structură de bază pentru REST API.

Într-o aplicație REST API, Controller-ele sunt utilizate pentru a defini metodele care gestionează cererile HTTP (GET, POST, PUT, DELETE). Fără controllere, se poate utiliza un model minimal bazat pe **Endpoints** (MapGet, MapPost etc.), dar acesta nu este ideal pentru proiecte complexe.

Controller-ele oferă o separare clară a logicii aplicației, ceea ce face proiectul mai ușor de înțeles și întreținut. Astfel, se pot organiza **endpoint-urile** pe mai multe controllere (de exemplu: ***ArticlesController***, ***ProductsController, etc.***).

Identity utilizează controlere personalizate sau extensii în controlerele API-ului pentru gestionarea utilizatorilor (înregistrare, autentificare, roluri). Controller-ele permit crearea **endpoint-urilor** clare pentru gestionarea utilizatorilor și a autentificării.

Un **Endpoint** este un punct final de comunicare în cadrul unui API, utilizat pentru a interacționa cu aplicația. Într-un API RESTful, un **endpoint** este asociat cu o **adresă URL** specifică, care permite clienților să trimită cereri și să primească răspunsuri.

Elementele cheie ale unui endpoint:

➢ **URL (Uniform Resource Locator):**
- Adresa web care identifică resursa
(de exemplu: https://example.com/api/products)

- **Metodă HTTP (HTTP Verb):**
  - Specifică tipul de operațiune care o să aibă loc asupra resursei.

    Metodele comune includ:
    - **GET** – afișează informații
    - **POST** – creează o resursă
    - **PUT** – actualizează o resursă
    - **DELETE** – șterge o resursă

- **Resursa asociată:**
  - Resursa este entitatea cu care se interacționează (de exemplu: articole, produse, utilizatori, comenzi) **Exemplu: /products** indică faptul că endpoint-ul gestionează resurse de tip produs.

- **Parametrii opționali:**
  - Pot include parametri de cale (**/products/{id}**), parametri de query (**?page=1**) sau corpul cererii (body) pentru operațiuni cum ar fi POST sau PUT.

Exemplu de endpoint pentru obținerea detaliilor unui produs specific:
- URL: /api/products/{id}
- Metodă HTTP - GET
- Exemplu: /api/products/5 (pentru produsul cu Id-ul 5).

Cum funcționează?
- Clientul trimite o cerere HTTP către endpoint-ul specificat;
- Serverul procesează cererea (apelând metoda asociată din Controller);
- Serverul returnează un răspuns HTTP către client;

Introducere în REST API

Ce este REST API?

REST (Representational State Transfer) este un stil arhitectural utilizat pentru comunicarea între client și server, folosind HTTP. În locul returnării unor View-uri (HTML), așa cum am procedat până în acest moment, un **REST API** returnează date în format ***JSON*** sau ***XML***.

Importanța utilizării unui REST API?

- Separare completă între client și server
- Poate fi consumat de diverse aplicații: web, mobile, IoT
- Este scalabil și reutilizabil

Sistemul de fișiere

Structura fișierelor în ASP.NET Core REST API este simplificată:

- **Controllers –** conțin metodele pentru endpoint-uri.
- **Models –** definește entitățile și structura datelor
- **Data -** conține contextul bazei de date (DbContext)
- **wwwroot:** Poate fi folosit pentru fișiere statice dacă este necesar

Sistemul de Rutare

Rutarea în REST API folosește convențiile REST pentru gestionarea resurselor prin cereri HTTP specifice:

- **GET /api/products** – listează toate produsele
- **POST /api/products** – creează un produs nou
- **PUT /api/products/{id}** – actualizează un produs existent
- **DELETE /api/products/{id}** - șterge un produs existent

Exemplu de implementare:

```
[Route("api/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
    [HttpGet]
    public IActionResult GetProducts() { ... }

    [HttpPost]
    public IActionResult CreateProduct(Product product) {
... }
}
```

Configurarea Identity Framework

Pentru a adăuga gestionarea utilizatorilor și rolurilor, se configurează ***ASP.NET Core Identity***:

- Se adaugă pachetele:
  Microsoft.AspNetCore.Identity.EntityFrameworkCore
  Microsoft.EntityFrameworkCore.SqlServer
  Microsoft.EntityFrameworkCore.Tools

Se vor selecta pachetele compatibile cu versiunea **.NET 9**, utilizându-se versiunea 9.0.9.

- Se configurează serviciile Identity în **Program.cs**

```
builder.Services.AddDbContext<AppDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionStr
ing("DefaultConnection")));
builder.Services.AddIdentity<IdentityUser, IdentityRole>()
    .AddEntityFrameworkStores<AppDbContext>()
            .AddDefaultTokenProviders();
```

- Se extinde contextul bazei de date pentru a include Identity. Contextul se creează într-un folder Data (acest folder se creează și el):

```
public class AppDbContext : IdentityDbContext<IdentityUser>
{
    public AppDbContext(DbContextOptions<AppDbContext>
options) : base(options) { }
}
```

- Se creează o bază de date și se adaugă stringul de conexiune în **appsettings.json**

Structura recomandă pentru proiectele REST API:

```
/ApiProject
│
├── /Controllers
│   ├── ProductsController.cs
│   ├── ArticlesController.cs
│
├── /Models
│   ├── Product.cs
│   ├── Article.cs
│
├── /Data
│   ├── AppDbContext.cs
│
├── /Services
│   ├── ProductService.cs
│
├── Program.cs
├── appsettings.json
```

- Se adaugă migrațiile și se actualizează baza de date:

```
Add-Migration AddIdentity
Update-Database
```

Implementarea CRUD pentru ArticlesController

Clasa Article.cs

```
public class Article
{
    [Key]
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public DateTime Date { get; set; }

    // Relatia cu Category
    public int? CategoryId { get; set; }
    public virtual Category? Category { get; set; }

    // Relatia cu IdentityUser
    public string? UserId { get; set; }
    public virtual IdentityUser? User { get; set; }
}
```

Clasa Category.cs

```
public class Category
{
    [Key]
    public int Id { get; set; }

    [Required(ErrorMessage = "Categoria este obligatorie")]
    public string CategoryName { get; set; }

    // Relatia 1:M cu Article

    [JsonIgnore]
    public ICollection<Article> Articles { get; set; } =
new List<Article>();
}
```

Utilizarea **[JsonIgnore]** pe proprietatea virtuală este necesară în **REST API** pentru a preveni ciclurile de serializare care apar din cauza relațiilor **bidirecționale** între entități. Această problemă nu apare, de obicei, în ASP.NET Core MVC, deoarece MVC generează HTML, nu JSON.

Relațiile din EF Core (cum este relația între Category și Article) sunt, de regulă, bidirecționale. **Category** are o colecție de articole (Articles). **Article** are o referință la categoria sa (Category). În timpul serializării JSON, serializatorul parcurge aceste relații și încearcă să includă toate obiectele și sub-obiectele asociate, ducând la o recursivitate infinită.

Contextul bazei de date:

```
  public class AppDbContext : IdentityDbContext<IdentityUser>
  {
  public AppDbContext(DbContextOptions<AppDbContext> options)
: base(options) { }


    public DbSet<Article> Articles { get; set; }
    public DbSet<Category> Categories { get; set; }

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Configurarea relatiilor
        modelBuilder.Entity<Article>()
            .HasOne(a => a.Category)
            .WithMany(c => c.Articles)
            .HasForeignKey(a => a.CategoryId);

        modelBuilder.Entity<Article>()
            .HasOne(a => a.User)
            .WithMany()
            .HasForeignKey(a => a.UserId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

Crearea Controller-ului împreună cu adnotările:

```
[Route("api/[controller]")]
[ApiController]
public class ArticleController : ControllerBase
{
	private readonly AppDbContext db;

	public ArticleController(AppDbContext context)
	{
		db = context;
	}

	…
}
```

Adnotările `[Route("api/[controller]")]` și `[ApiController]` sunt utilizate în ASP.NET Core pentru a configura comportamentul unui Controller și pentru a defini rutele care vor fi utilizate pentru endpoint-urile sale.

`[Route("api/[controller]")]` – această adnotare definește ruta de bază pentru toate metodele dintr-un Controller. În acest caz, ruta este **api/[controller]**, **[controller]** fiind un token care va fi înlocuit automat cu numele Controller-ului, fără sufixul "Controller". Dacă numele Controller-ului este **ArticlesController**, ruta de bază devine **api/articles.**

`[ApiController]` – această adnotare configurează automat comportamentul specific unui API Controller în ASP.NET Core, furnizând funcționalități precum:

- Validarea automată a modelului **(ModelState)**;
- Legarea automată a datelor **(Model Binding) –** ASP.NET Core știe automat de unde să preia datele din cerere (query string, body, etc.), pe baza tipului parametrilor și a adnotărilor utilizate (exemplu: [FromBody], [FromQuery]);
- Rute mai consistente – asigură configurarea corectă a rutelor și combinarea acestora cu metodele HTTP (exemplu: [HttpGet], [HttpPost]);

Afișarea unui articol, împreună cu categoria din care face parte și cu utilizatorul care a postat articolul respectiv

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Article>>>
GetArticles()
{
    var articles = await db.Articles
        .Include(a => a.Category)
        .Include(a => a.User)
        .ToListAsync();

     if (articles == null || articles.Count == 0)
          return NotFound(); // returnează 404 daca nu
sunt articole

     return Ok(articles); // returneaza OK si lista de
articole
}
```

Explicații:

Metoda este **publică,** modificatorul de acces indicând faptul că metoda este accesibilă de oriunde. În cazul unui API, metodele publice dintr-un Controller definesc endpoint-urile accesibile din exterior.

Async indică faptul că metoda este asincronă. O metodă marcată cu async nu blochează alte operații existente, ceea ce este util mai ales pentru apelurile la bazele de date sau alte resurse externe. Aceasta abordare îmbunătățește performanța și scalabilitatea API-ului, permițând serverului să gestioneze alte cereri în timpul așteptării. O metodă asincronă returnează un **Task**, care reprezintă o operație ce se va finaliza în viitor, ceea ce înseamnă că răspunsul final al metodei va fi disponibil după ce operațiunea este completă.

Gestionarea Utilizatorilor

Configurarea autentificării în API

Într-un API RESTful cu **Identity**, utilizatorii se autentifică folosind **JSON Web Tokens (JWT)**.

PASUL 1:

Se instalează pachetul ***Microsoft.AspNetCore.Authentication.JwtBearer*** și ***Swashbuckle.AspNetCore***

Cum funcționează JWT în autentificare:

- Atunci când un user se loghează, clientul (browserul) trimite datele de autentificare (email și parolă) către server. Serverul validează datele de autentificare, generează un JWT conținând informațiile utilizatorului și îl returnează clientului.

- Clientul stochează token-ul JWT (de obicei în local storage sau cookies), iar la fiecare cerere către API clientul trimite JWT-ul în antetul Authorization: Bearer <token>

- Serverul validează token-ul (verifică semnătura pentru autenticitate și dacă token-ul este încă valid (nu a expirat). Dacă token-ul este valid, serverul procesează cererea în numele userului care posedă acel token.

PASUL 2:

Se configurează JWT în Program.cs:

```
builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

builder.Services.AddOpenApi();

builder.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 = builder.Configuration["Jwt:Issuer"],
        ValidAudience =
builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configu
ration["Jwt:Key"]))
    };
    });

// Adaugarea posibilitati de a folosi bearer token in
swagger
// Se utilizeaza pentru a avea posibilitatea de a adauga un
token in swagger
builder.Services.AddSwaggerGen(option =>
{
    option.AddSecurityDefinition("Bearer", new
OpenApiSecurityScheme
    {
        In = ParameterLocation.Header,
```

```
            Description = "Please enter a valid token",
            Name = "Authorization",
            Type = SecuritySchemeType.Http,
            BearerFormat = "JWT",
            Scheme = "Bearer"
        });
        option.AddSecurityRequirement(new
OpenApiSecurityRequirement
        {
            {
                new OpenApiSecurityScheme
                {
                    Reference = new OpenApiReference
                    {
                        Type=ReferenceType.SecurityScheme,
                        Id="Bearer"
                    }
                },
                new string[]{}
            }
         });
});
```

PASUL 3:

Se adaugă cheile JWT în appsettings.json:

```
"Jwt": {
  "Key":
"tdYPFEUoqH6xIEHUMtTED4UWeo5qXGbJV0xxccNd28kGqkQ1U4wgWRmwpT
Drg37E",
  "Issuer": "http://localhost",
  "Audience": "http://localhost"
},
```

Cheile secrete se generează folosind tool-uri în linia de comandă.

În cazul de față s-a utilizat o aplicație externă (nefiind o modalitate potrivită în producție) - https://generate-random.org/encryption-key-generator

Key - este o cheie secretă utilizată pentru a semna și valida JWT-urile. Trebuie să fie o cheie puternică, complexă, pentru a preveni problemele de securitate.

Issuer – reprezintă entitatea care emite token-ul, adică serverul sau aplicația curentă. Trebuie să aibă o valoare unică (de exemplu: numele domeniului aplicației: https://myapp.com). Aceasta este verificată în timpul validării token-ului.

Audience - indică pentru cine este destinat token-ul (aplicația sau clienții care îl vor folosi). De obicei, este o valoare precum: https://myapi.com (adresa API-ului) sau un identificator al aplicației client.

OBS!

Cheile nu trebuie păstrate în codul sursă. Este indicată folosirea unui sistem de gestionare a cheilor (de exemplu.: Azure Key Vault, AWS Secrets Manager sau dotnet Secret Manager). Acestea se pot seta și în variabile de mediu.

PASUL 4:

Se configurează autentificarea în pipeline-ul middleware (în Program.cs)

```
app.UseAuthentication();
app.UseAuthorization();
```

PASUL 5:

Crearea unui endpoint pentru autentificare:

```
[Route("api/[controller]")]
[ApiController]
public class AccountController : ControllerBase
{
    private readonly UserManager<IdentityUser>
_userManager;
    private readonly IConfiguration _configuration;

    public AccountController(UserManager<IdentityUser>
userManager, IConfiguration configuration)
    {
        _userManager = userManager;
        _configuration = configuration;
    }

    [HttpPost("register")]
    public async Task<IActionResult> Register([FromBody]
AuthModel model)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        var user = new IdentityUser { UserName =
model.Email, Email = model.Email };
        var result = await _userManager.CreateAsync(user,
model.Password);

        if (!result.Succeeded)
            return BadRequest(result.Errors);

        return Ok(new { Message = "User registered
successfully" });
    }
```

```
    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody]
AuthModel model)
    {
        var user = await
_userManager.FindByEmailAsync(model.Email);
        if (user == null || !await
_userManager.CheckPasswordAsync(user, model.Password))
            return Unauthorized(new { Message = "Invalid
credentials" });

        var authClaims = new[]
        {
        new Claim(ClaimTypes.Name, user.UserName),
        new Claim(ClaimTypes.NameIdentifier, user.Id),
        new Claim(JwtRegisteredClaimNames.Jti,
Guid.NewGuid().ToString())
    };

        var token = new JwtSecurityToken(
            issuer: _configuration["Jwt:Issuer"],
            audience: _configuration["Jwt:Audience"],
            expires: DateTime.Now.AddHours(1),
            claims: authClaims,
            signingCredentials: new SigningCredentials(
                new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration[
"Jwt:Key"])),
                SecurityAlgorithms.HmacSha256)
        );

        return Ok(new { Token = new
JwtSecurityTokenHandler().WriteToken(token) });
    }
}

public class AuthModel
{
    public string Email { get; set; }
    public string Password { get; set; }
}
```

PASUL 6:

Implementarea ArticlesController:

```
[Route("api/[controller]")]
 [ApiController]
 //[Route("all-articles")]
 public class ArticleController : ControllerBase
 {
     private readonly AppDbContext db;

     public ArticleController(AppDbContext context)
     {
         db = context;
     }

     [HttpGet]
     public async Task<ActionResult<IEnumerable<Article>>>
Index()
     {
         var articles = await db.Articles
             .Include(a => a.Category)
             .Include(a => a.User)
             .ToListAsync();

         if (articles == null || articles.Count == 0)
             return NotFound(); // returnează 404 daca nu
sunt articole

         return Ok(articles); // returneaza OK si lista de
articole
     }


     [HttpPost]
     public async Task<IActionResult> New([FromBody]
Article article)
     {
         if (!ModelState.IsValid)
             return BadRequest(ModelState);

         // preluam Id-ul utilizatorului care posteaza
articolul

         var userId =
User.FindFirstValue(ClaimTypes.NameIdentifier);
```

```
            if(userId is null)
            {
                return Unauthorized(new
                {
                    Message = "User not authenticated"
                });
            }

            article.UserId = userId;

            var category = await
db.Categories.FindAsync(article.CategoryId);

            if (category == null)
                return NotFound(new { Message = "Category not
found" });

            db.Articles.Add(article);
            await db.SaveChangesAsync();

            return CreatedAtAction(nameof(Index), new { id =
article.Id }, article);
        }


        [HttpPut("{id}")]
        public async Task<IActionResult> Edit(int id,
[FromBody] Article updatedArticle)
        {
            if (id != updatedArticle.Id)
            {
                return BadRequest(new { Message = "Article Id
mismatch" });
            }

            var article = await db.Articles.FindAsync(id);
            if (article == null)
            {
                return NotFound(new { Message = "Article not
found" });
            }

            var userId =
User.FindFirstValue(ClaimTypes.NameIdentifier);
```

```
            if (userId is null)
            {
                return Unauthorized(new
                {
                    Message = "User not authenticated"
                });
            }

            if (article.UserId != userId)
            {
                return Unauthorized(new { Message = "You are
not allowed to edit this article" });
            }


            // se actualizeaza campurile articolului
            article.Title = updatedArticle.Title;
            article.Content = updatedArticle.Content;
            article.Date = updatedArticle.Date;
            article.CategoryId = updatedArticle.CategoryId;

            try
            {
                await db.SaveChangesAsync();
            }
            catch (DbUpdateException ex)
            {
                return BadRequest(new { Message = "An error
occurred while updating the article", Error = ex.Message
});
            }

            return Ok(new { Message = "Article updated
successfully" });
        }


        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
            var article = await db.Articles.FindAsync(id);
            if (article == null)
            {
                return NotFound(new { Message = "Article not
found" });
            }

            var userId =
User.FindFirstValue(ClaimTypes.NameIdentifier);
```

```
            if (userId is null)
            {
                return Unauthorized(new
                {
                    Message = "User not authenticated"
                });
            }

            if (article.UserId != userId)
            {
                return Unauthorized(new { Message = "You are
not allowed to delete this article" });
            }

            db.Articles.Remove(article);

            try
            {
                await db.SaveChangesAsync();
            }
            catch (DbUpdateException ex)
            {
                return BadRequest(new { Message = "An error
occurred while deleting the article", Error = ex.Message
});
            }

            return Ok(new { Message = "Article deleted
successfully" });
        }
}
```

Pentru ca aplicația să se deschidă în browser, se accesează Properties → launchSettings.json → se modifică valoarea parametrului **launchBrowser** în True.

Swagger va fi accesibil la: https://localhost:7020/swagger/