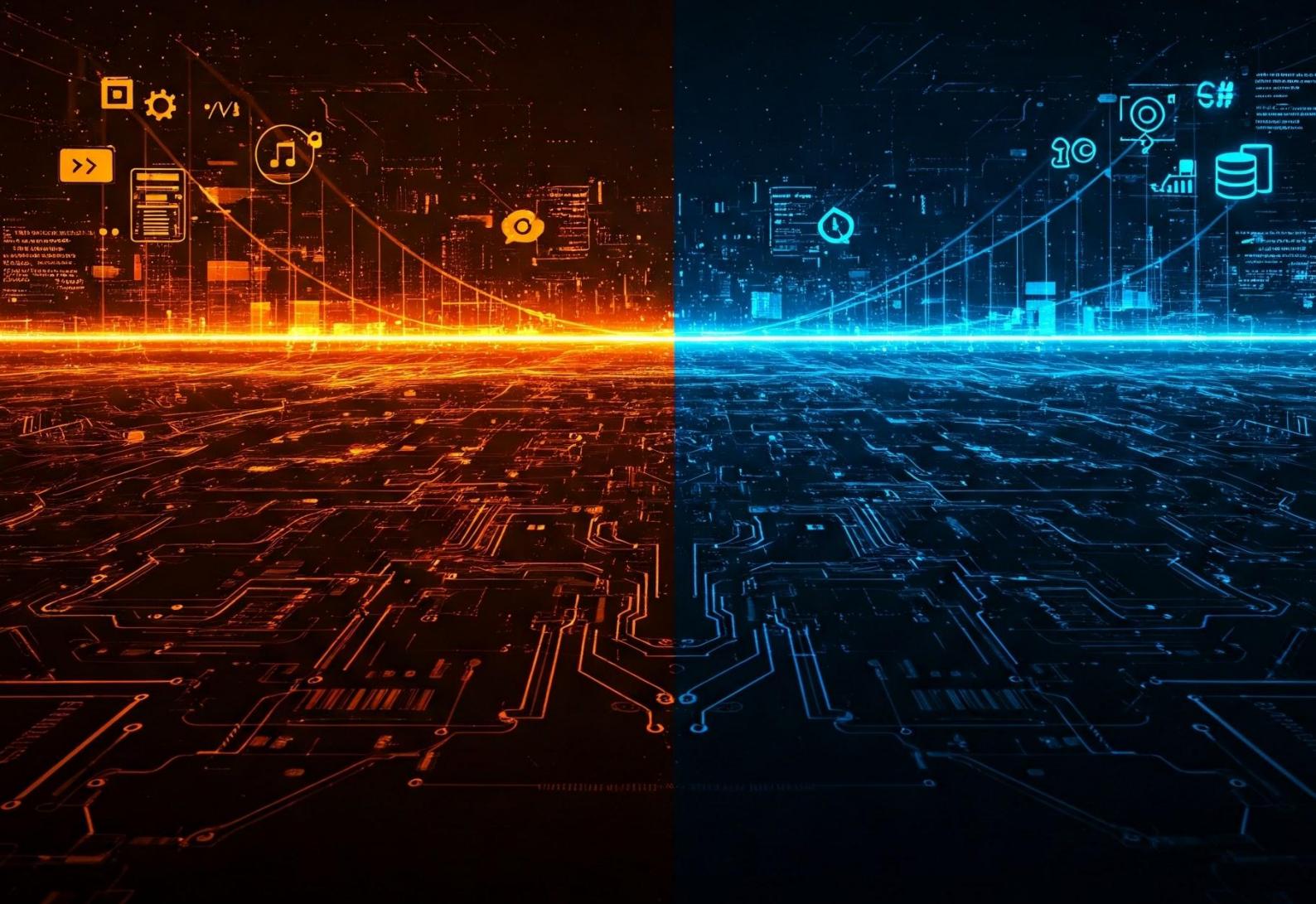


ASP.NET Core Web API: Praktični vodič za Angular developere

Zoran Bošnjak



Predgovor

Dobro došli! Ovaj vodič nastao je s ciljem da **Angular developerima** omogući brz i praktičan uvod u izradu **.NET Web API** aplikacija. Ako već vladate Angularom i TypeScriptom, vjerojatno želite proširiti svoje vještine i savladati backend tehnologije. .NET Core i ASP.NET Core Web API moderna su Microsoft rješenja za izradu web-aplikacija i API-ja. Kroz ovaj ćemo te vodič provesti korak po korak kroz osnove C# programskog jezika i ASP.NET Core Web API-ja, s naglaskom na **praktične primjere koda** umjesto opširne teorije.

Vodič je strukturiran prema planu učenja raspoređenom na 10 tjedana, no možete napredovati i brže ili sporije prema vlastitom ritmu. Svako poglavlje odgovara jednom logičnom koraku u učenju – od postavljanja okruženja, preko upoznavanja C# sintakse i objektno orientiranog programiranja, do izrade prvog API-ja, povezivanja s bazom podataka, implementacije sigurnosti (JWT autentifikacija) i završnog mini-projekta koji povezuje sve naučeno.

Kako koristiti ovaj vodič? Preporuka je da čitate poglavlje po poglavlje i isprobate priloženi kod na svom računalu. Ne brinite, kodni primjeri su zamišljeni tako da budu razumljivi i kratki, a možete ih pokretati uz minimalne preduvjete (dovoljno je imati instaliran .NET SDK i omiljeni editor poput Visual Studio Code-a ili Visual Studija). Cilj je da *što prije stvarno programirate* i eksperimentirate s kodom. Teoriju smo sveli na osnovne koncepte koje odmah primjenjujemo u praksi.

Na kraju ovog vodiča, trebali bi biti sposobni samostalno postaviti i izgraditi jednostavan Web API na .NET Core platformi, razumjeti ključne koncepte kao što su **kontroleri**, **rute**, **HTTP metode**, **CRUD operacije**, **dependency injection**, **Entity Framework Core** za rad s bazom, konfiguracija aplikacije i **JWT autentifikacija** za sigurnost. To je sjajna osnova na kojoj se može nastaviti nadograđivati svoje znanje.

Spremni? U sljedećem dijelu nalazi se **sadržaj** vodiča s popisom poglavlja. Krenimo redom i zaronimo u svijet .NET Web API-ja! Sretno kodiranje!

Zoran

Ožujak 2025.

Sadržaj

Predgovor	1
Tjedan 1: Postavljanje okruženja i osnove C# sintakse	5
Postavljanje .NET okruženja	6
Osnove C# sintakse	7
Zaključak poglavlja	9
Tjedan 2: Objektno orijentirano programiranje (OOP) u C#	10
Klase i objekti u C#	11
Nasljeđivanje (Inheritance)	13
Value tipovi i referentni tipovi	15
Zaključak poglavlja	16
Tjedan 3: .NET Core i osnove Web API-ja	17
Što je .NET Core i ASP.NET Core?	18
Struktura ASP.NET Core Web API projekta	18
Kontroleri i rute	22
Što ovdje radimo?	22
Metode u kontroleru – HTTP rute	23
Vraćanje objekata umjesto plain stringova	25
Što se ovdje događa?	25
Kako izgleda odgovor?	26
Zaključak	26
Pokretanje i testiranje API-ja	27
Testiranje API-ja preko Swaggera	28
Životni ciklus HTTP zahtjeva u ASP.NET Core Web API	29
Zaključak poglavlja	31
Tjedan 4: Izrada prvog Web API endpointa (CRUD u memoriji)	32
Definiranje kontrolera i rute	33
Isprobavanje CRUD endpointa	36
JSON serijalizacija i povratne vrijednosti	37
Praktični savjeti za vraćanje odgovora	37
Koji statusni kod koristiti?	37
Zaključak poglavlja	38

Tjedan 5: Dependency Injection u .NET Core-u	39
Što je Dependency Injection?.....	40
Registracija servisa u ASP.NET Core.....	40
Zašto je DI koristan?	43
Napomena o lifetime: Transient vs Scoped vs Singleton.....	44
Zaključak poglavlja	45
Tjedan 6: Integracija baze podataka s Entity Framework Core	46
Uvod u Entity Framework Core	47
Postavljanje EF Core u projekt.....	48
Definiranje DbContext-a i modela	49
Registracija DbContext-a u DI i konfiguracija konekcije	50
Ažuriranje kontrolera na EF Core	51
Što se promjenilo u kodu?	53
Migracije i kreiranje baze.....	55
Testiranje s bazom	55
Još malo o EF Core	56
Zaključak poglavlja	58
Tjedan 7: Konfiguracija aplikacije i appsettings.json	59
appsettings.json i konfiguracijski sustav.....	60
Čitanje konfiguracije u kodu.....	61
Korištenje konfiguracije u praksi.....	62
Primjena na naš projekt	63
Još o appsettings.Development.json.....	63
Zaključak poglavlja	64
Tjedan 8: Osnove autentifikacije i sigurnosti Web API-ja	65
Autentifikacija vs. autorizacija.....	66
Postavljanje JWT autentifikacije u ASP.NET Core	67
Objašnjenje konfiguracije JWT autentifikacije.....	68
Integracija autentifikacije u middleware pipeline.....	69
Generiranje JWT tokena.....	69
Zaštita endpointa pomoću [Authorize].....	73
Sigurnosni savjeti pri radu s JWT autentifikacijom	74
Autorizacija po ulogama (Role Based Authorization)	76
Zaključak poglavlja	77
Tjedan 9: Izrada mini projekta: kompletan Web API	78

Postavljanje projekta i modela	79
Kreiranje DbContext za bazu	81
Task Controller (CRUD logika).....	83
Autentikacija (ponovno) i korisnici.....	85
Swagger UI i testiranje	86
Zaključak poglavlja	88
Tjedan 10: Pregled, dorada i sljedeći koraci.....	89
Kratki pregled naučenog	90
Dorada i najbolje prakse	91
Samoprovjera znanja	92
Sljedeći koraci (što dalje učiti).....	93
Provjerite stečeno znanje u praksi – napredniji WebApi.....	95
Opis zadatka	96
Zaključna riječ	98
Dodatak: 100 najčešćih pitanja za junior i mid-level developere.....	99
Osnove C# programskog jezika	100
Web API u .NET-u (ASP.NET Core Web API).....	101
Entity Framework Core i rad s bazom podataka	102
LINQ upiti i obrada podataka	102
Dependency Injection i arhitektura aplikacije	103
Osnove performansi i sigurnosti u backend aplikacijama	104

Tjedan 1: Postavljanje okruženja i osnove C# sintakse

U ovom poglavlju uspostavit ćemo razvojno okruženje za .NET i upoznati se s osnovama C# sintakse. Krenut ćemo s jednostavnim *'Hello World'* programom i istražiti ključne koncepte jezika. Cilj je da se osjećamo ugodno s C# kodom prije nego što priđemo na razvoj Web API-ja.

Postavljanje .NET okruženja

Prvi korak je instalacija potrebnih alata. Trebat će nam **.NET SDK** (Software Development Kit) i neki uređivač koda ili IDE. Ako već koristite Visual Studio Code za Angular, možete nastaviti s njim (uz C# ekstenziju), ili možete instalirati **Visual Studio** Community izdanje koje ima sve potrebno integrirano. Puna verzija Visual Studio je uobičajena za korištenje i mnogo češće se koristi od VS Codea za .NET razvoj. Nakon instalacije .NET SDK-a, možete provjeriti uspješnu instalaciju otvaranjem terminala/Command Prompt-a i pokretanjem:

```
dotnet --version
```

Ova naredba bi trebala ispisati broj verzije .NET-a koji je instaliran, npr. 9.0.x. Također, .NET dolazi s alatima za komandnu liniju. Za brzi test, pokušajte kreirati i pokrenuti najjednostavniji projekt:

```
dotnet new console -o HelloWorld  
cd HelloWorld  
dotnet run
```

Trebali bi dobiti ispis "Hello World!" u konzoli. Time smo potvrdili da okruženje radi i spremni smo za učenje C# sintakse.

Osnove C# sintakse

C# je objektno orijentirani jezik koji sintaksom pomalo podsjeća na JavaScript/TypeScript. Korištenje **vitičastih zagrada** za blokove koda, **točka-zarez** za kraj linije, if/else uvjete, for petlje - sve su to koncepti koji će vam djelovati poznato.

Evo jednog primjera jednostavnog C# programa koji ispisuje poruku i nekoliko brojeva:

```
using System;

namespace PrimerApp {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Pozdrav iz C#!"); // Ispis na konzolu

            // Primjer jednostavne petlje
            for (int i = 1; i <= 3; i++) {
                Console.WriteLine($"Broj: {i}");
            }
        }
    }
}
```

Objašnjenje: Ovdje vidimo osnovnu strukturu C# konzolne aplikacije. `using System;` omogućuje korištenje klasa iz System namespacea, poput Console. Kod je organiziran unutar **namespace** (ovdje PrimerApp), a imamo i definiciju klase Program s glavnom metodom Main.

Metoda Main je početna točka .NET konzolne aplikacije. U njoj koristimo Console.WriteLine za ispis. Petlja for je praktički ista kao u JavaScriptu/TypeScriptu, uz razliku što je `int i` jasno tipiziran kao cijeli broj i što string interpolaciju radimo sa `$"{i}"` slično kao u typescript (doduše TypeScript koristi `` ${}`` s backtickovima, dok C# koristi `"..."`).

Ako pokrenete ovaj program (npr. spremite ga u datoteku **Program.cs** unutar konzolne aplikacije i izvršite `dotnet run`), dobiti ćete ispis:

Pozdrav iz C#!

Broj: 1

Broj: 2

Broj: 3

Primijetimo nekoliko stvari:

- C# je **statički tipiziran** jezik, što znači da svaka varijabla ima fiksni tip (`int`, `string`, `bool`, itd.). Međutim, kompjuter je dovoljno pametan da u nekim slučajevima zaključi tip umjesto nas (npr. možeš koristiti ključnu riječ `var` za lokalnu varijablu pa će kompjuter odrediti tip na osnovu dodijeljene vrijednosti).
- Sintaksa uvjetnih naredbi (`if (uvjet) { ... } else { ... }`) i petlji (`for`, `while`, `foreach`) vrlo je slična onome što već koristite u JavaScriptu/TypeScriptu. Na primjer:

```
if (DateTime.Now.Hour < 12) {
    Console.WriteLine("Dobro jutro!");
} else {
    Console.WriteLine("Dobar dan ili dobra večer!");
}
```

- C# razlikuje **velika/mala slova** (case-sensitive) isto kao i JS/TS, pa npr. `name` i `Name` ne znače isto.

Za sada se držimo osnovnih tipova (`int`, `string`, `bool`, `float`, itd.) i konstrukcija. Napravite par mini vježbi: recimo, napišite kod koji provjerava je li broj paran ili neparan (uz `if`), ili listu od par stringova koje se kroz `foreach` petlju ispisuju. Tako ćete se naviknuti na rad s jezikom. U ovom tjednu izbjegavamo naprednije teme; nije potrebno odmah ulaziti u LINQ, delegate, evenete ili slične koncepte (iako ih ne škodi barem provjeriti). Također, iako je C# objektno orientiran, detalje OOP-a ostavit ćemo za sljedeće poglavje. Bitno je da se savlada sintaksom i osnovnim konstrukcijama - dalje će sve nadogradnje biti jednostavnije.



Napomena: Ako se koristi Visual Studio ili VS Code, primijetit ćete da imaju IntelliSense (pametno automatsko dovršavanje) i druge pomoćne alate. Iskoristite ih - npr. kada krenete pisati `Console.Wr...`, editor će ponuditi `WriteLine`. To olakšava pisanje koda i učenje dostupnih metoda.

Zaključak poglavlja

Kako bi lakše pratili kasnije teme, preporučujemo da istražite i eksperimentirate s osnovama:

- **Varijable i tipovi podataka**
 - int, double, bool, string, char
 - Deklaracija i inicijalizacija varijabli (int broj = 10;)
- **Kontrola toka programa**
 - if-else uvjetne naredbe
 - switch-case
 - Petlje: for, while, do-while, foreach
- **Kolekcije i rad s podacima**
 - Polja (int[] brojevi = {1, 2, 3};)
 - Liste (List<int> brojevi = new List<int>();)
 - Dictionary (Dictionary<string, int> ocjene = new Dictionary<string, int>();)
- **Metode**
 - Definiranje i pozivanje metoda
 - Parametri i povratne vrijednosti

Ako su vam ove stvari nove, isprobajte ih kroz kratke primjere kako bi stekli osnovno razumijevanje sintakse. C# je vrlo čitljiv jezik, a što više se vježba, brže će se osjećati ugodno u radu s njim.

Kada steknemo temelje, možemo prijeći na dublje koncepte. **U sljedećem poglavlju upoznat ćemo se s objektno orijentiranim programiranjem u C#-u, što će nam pomoći da bolje razumijemo strukturu Web API-ja.**

Tjedan 2: Objektno orijentirano programiranje (OOP) u C#

Cilj ovog poglavlja je razumjeti kako C# implementira objektno orijentirano programiranje i uočiti sličnosti s TypeScriptom. Naučit ćemo kako definirati klase, instancirati objekte, koristiti svojstva i metode te kroz praktične primjere objasniti ključne koncepte poput *enkapsulacije* i *nasljeđivanja*.

Klase i objekti u C#

U C# (kao i u TypeScriptu), klase su nacrti (template) prema kojima stvaramo objekte. Ako ste u Angularu pravili klase za modele podataka ili servise, koncept vam neće biti stran. Definirajmo jednostavnu klasu **Product** da bismo ilustrirali osnove:

```
public class Product {

    // Svojstva (properties)
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    // Konstruktor klase
    public Product(int id, string name, decimal price) {
        Id = id;
        Name = name;
        Price = price;
    }

    // Metoda koju možemo pozvati
    public void PrintInfo() {
        Console.WriteLine($"Proizvod: {Name}, cijena: {Price} eur");
    }
}
```

Objašnjenje:

- Klasa **Product** ima tri svojstva: **Id**, **Name** i **Price**. Kada pišemo `public int Id { get; set; }`, to znači da se vrijednost može čitati i mijenjati, a C# automatski upravlja pohranom podataka iza kulisa. U TypeScriptu bismo imali nešto slično s javnim poljem ili get/set metodama.
- Konstruktor je posebna metoda s istim imenom kao i klasa (Product). Poziva se kada stvaramo novi objekt i omogućuje nam da odmah postavimo Id, Name i Price.
- PrintInfo je metoda koja ispisuje podatke o proizvodu. Koristimo `Console.WriteLine` za ispis imena i cijene proizvoda, uz string interpolaciju (`($"Tekst {varijabla}")`) kako bi na jednostavan način kombinirali varijable i tekst. Cijena je tipa `decimal`, jer je taj tip pogodan za financijske iznose, ali mogli smo koristiti i `float` ili `double` za općenite decimalne vrijednosti.

Sada, kako koristiti ovu klasu? Trebamo **kreirati instance** (objekte) i pozivati metode/svojstva:

```
// Instanciranje objekta klase Product
Product p1 = new Product(1, "Olovka", 3.5m);
Product p2 = new Product(2, "Bilježnica", 15.0m);

// Pristup svojstvima
Console.WriteLine(p1.Name); // ispisuje "Olovka"
Console.WriteLine($"Proizvod {p2.Id}: {p2.Name}, cijena {p2.Price} eur");

// Pozivanje metode
p1.PrintInfo(); // ispisuje "Proizvod: Olovka, cijena: 3.5 eur"
```

Objašnjenje:

Kreirali smo dva proizvoda koristeći `new Product(...)`. Primijetite slovo **m** uz decimalne vrijednosti – ono označava da se radi o **decimal** tipu (m dolazi od "money"/decimal, slično kao što **f** označava **float**).

Nakon toga, ispisujemo neka svojstva proizvoda i pozivamo metodu **PrintInfo()** na prvom proizvodu, koja prikazuje informacije o njemu

Slično kao u Angularu, kada imate klasu za model podataka, bolje je proslijediti cijelu instancu klase umjesto da pojedinačno šaljete **id**, **ime** i **cijenu**.

Također, pomoću **modifikatora pristupa (public, private, itd.)** određujemo što je dostupno izvana, a što ostaje interno u klasi. U našem primjeru, sva svojstva su **public** radi jednostavnosti, ali u stvarnim scenarijima neke bismo informacije mogli ostaviti **privatnima** i omogućiti pristup samo putem metoda.

 Ključni OOP koncept ovdje je **enkapsulacija**: klasa Product *kapsulira* (objedinjuje) više vrijednosti (Id, Name, Price) i ponašanja (PrintInfo) u jednu logičku cjelinu.

Nasljeđivanje (Inheritance)

C# podržava **nasljeđivanje klasa**, što znači da jedna klasa može **naslijediti** drugu i preuzeti njezina svojstva i metode. Ovo je slično **extends** konceptu u TypeScriptu.

Evo brzog primjera: recimo da imamo klasu **Product**, a želimo dodati specifične karakteristike za električne uređaje. Možemo napraviti podklasu **ElectronicProduct**, koja nasljeđuje **Product** i dodaje novo svojstvo, primjerice **trajanje baterije**:

```
// Podklasa ElectronicProduct nasljeđuje klasu Product
public class ElectronicProduct : Product {
    public int BatteryLifeHours { get; set; }

    // Konstruktor podklase poziva konstruktor bazne klase sa base(...)
    public ElectronicProduct(int id, string name, decimal price,
int batteryLife)
        : base(id, name, price) {
        BatteryLifeHours = batteryLife;
    }

    // Možemo dodati nove metode ili svojstva specifična za
    // ElectronicProduct
    public void ShowBattery() {
        Console.WriteLine($"Baterija traje: {BatteryLifeHours} sati.");
    }
}
```

Koristimo nasljeđivanje na slijedeći način:

```
ElectronicProduct smartphone =
new ElectronicProduct(5, "Smartphone", 2500m, 24);

// "Smartphone" - svojstvo naslijeđeno iz Product
Console.WriteLine(smartphone.Name);

smartphone.ShowBattery();      // "Baterija traje: 24 sati."

// Možemo čak pozvati PrintInfo() definiranu u Product klasi
smartphone.PrintInfo();
```

Što se ovdje događa?

- `ElectronicProduct` nasljeđuje `Product`, što znači da automatski dobiva svojstva Id, Name, Price i metodu PrintInfo iz bazne klase, bez potrebe da ih ponovno definiramo. Ovo nam omogućuje ponovnu upotrebu koda i izbjegava duplikiranje.
- Konstruktor `ElectronicProduct` koristi `: base(...)` sintaksu kako bi proslijedio Id, Name i Price konstruktoru bazne klase `Product`. Na taj način, inicijalizaciju tih svojstava prepuštamo baznoj klasi, dok se `BatteryLifeHours` postavlja unutar konstruktora podklase.
- Instanca `smartphone` je istovremeno tipa `ElectronicProduct` i `Product`, budući da nasljeđivanje znači da je podklasa poseban slučaj bazne klase. **To znači da `ElectronicProduct` može biti korišten bilo gdje se očekuje `Product`.**
- Za primjer, ako imamo `List<Product>`, možemo u nju dodati instancu `ElectronicProduct`, jer je i dalje kompatibilna s `Product` tipom. Ovo omogućuje fleksibilnost kod rada s kolekcijama i metodama koje rade s baznim tipom.

Interface (sučelje)

Još jedan koncept, **interface (sučelje)**, također je bitan, ali o njemu ćemo praktično govoriti u poglavlju o Dependency Injection-u. Za sada je dovoljno znati da sučelje u C# slično kao u TS: definira skup metoda/svojstava bez implementacije, koje onda klasa može implementirati.

 **Napomena:** C# podržava samo jednostruko nasljeđivanje, što znači da klasa može naslijediti samo jednu drugu klasu. Za razliku od nekih drugih objektno orijentiranih jezika, višestruko nasljeđivanje nije moguće.

Kod prelaska na **Web API**, važno je razumjeti da **ASP.NET Core koristi nasljeđivanje**. **Kontroleri** u Web API-ju su klase koje često nasljeđuju **ControllerBase**, baznu klasu koja pruža osnovnu funkcionalnost za rukovanje HTTP zahtjevima.

Uskoro ćemo vidjeti kako se ovo koristi u praksi!

Value tipovi i referentni tipovi

U C#-u postoji važna razlika između **value** i **referentnih tipova**.

- **Value tipovi** (*int, double, bool, struct...*) **pohranjuju stvarnu vrijednost**. Kada ih dodijelimo drugoj varijabli ili proslijedimo metodi, **kopira se sama vrijednost**, što znači da svaka varijabla ima **svoju neovisnu kopiju podataka**. Promjene na jednoj varijabli **ne utječu** na drugu.
- **Referentni tipovi** (*klase, nizovi, objekti...*) **ne sadrže sami objekt**, već samo **referencu na njegovu lokaciju u memoriji**. Kada referentni tip dodijelimo drugoj varijabli, **kopira se samo referenca**, a ne sam objekt.
- To znači da **promjena na objektu preko jedne variabile automatski utječe i na sve druge variable koje referenciraju taj isti objekt**. Ovo je slično načinu na koji objekti funkcioniraju u **JavaScriptu** – prosljeđuju se **po referenci**, a ne kopiranjem stvarnih podataka.

Ova razlika je ključna za razumijevanje rada s podacima u C#-u, osobito kada radimo s metodama i kolekcijama.

```
int a = 5;
int b = a;
b = 6;
Console.WriteLine(a); // a je i dalje 5 (value tip kopiran)

Product prodA = new Product(10, "Test", 99m);
Product prodB = prodA;
prodB.Name = "Novo ime";

// ispisuje "Novo ime" (oba referiraju isti objekt)
Console.WriteLine(prodA.Name);
```

Zaključak poglavlja

U ovom poglavlju obradili smo temeljne koncepte objektno orijentiranog programiranja (OOP) u C#: kreiranje klasa, instanciranje objekata, nasljeđivanje i osnovna OOP pravila. Ako već poznajete OOP kroz Angular i TypeScript, ovo vam vjerojatno djeluje prilično jednostavno. Ako ne, preporučujem da napišete vlastitu klasu i kratki program koji je koristi – to će pomoći da bolje usvojite znanje.

Ako želite dodatno produbiti svoje razumijevanje OOP principa, preporučujem istraživanje **četiri temeljna stupa OOP-a**:

- **Enkapsulacija** – skrivanje unutarnjih detalja klase i izlaganje samo potrebnih podataka kroz javne metode.
- **Nasljeđivanje** – omogućuje jednoj klasi da preuzme ponašanje i svojstva druge klase.
- **Polimorfizam** – sposobnost metode da se ponaša drugačije ovisno o tipu objekta koji ju poziva.
- **Apstrakcija** – pojednostavljivanje složenih sustava skrivanjem detalja i prikazivanjem samo ključnih funkcionalnosti.

Za praktično učenje, možete pokušati:

Napraviti **jednostavnu OOP hijerarhiju**, npr. klasu **Vozilo**, a zatim podklase **Auto** i **Motocikl**.

Dodati metode koje prikazuju polimorfizam, npr. **virtualne metode** koje podklase mogu nadjačati.

Eksperimentirati s **interfejsima** kako biste vidjeli razliku između nasljeđivanja klasa i implementacije interfejsa.

Sljedeći korak je prelazak s konzolnih aplikacija na web aplikacije. Srećom, .NET Core značajno pojednostavljuje razvoj web API-ja. U idućem poglavlju prvo ćemo se upoznati s konceptima i struktukom ASP.NET Core Web API-ja, a zatim krenuti s implementacijom konkretnog API-ja.

Tjedan 3: .NET Core i osnove Web API-ja

Cilj ovog poglavlja je razumjeti što su .NET Core i ASP.NET Core Web API te kako izgleda osnovna struktura Web API projekta. Pokrenut ćemo prvi jednostavni Web API i analizirati kako HTTP zahtjev prolazi kroz aplikaciju – od rute do kontrolera i metode.

Što je .NET Core i ASP.NET Core?

.NET Core je modularna i višeplatformska verzija .NET-a, što znači da se aplikacije mogu pokretati na Windowsu, Linuxu i macOS-u.

ASP.NET Core je dio .NET Core platforme namijenjen razvoju web aplikacija i servisa, uključujući Web API-je (Application Programming Interface). Web API je zapravo backend aplikacija koja izlaže podatke i funkcionalnosti putem HTTP protokola, najčešće u obliku RESTful API-ja koji vraćaju JSON podatke.

Kao Angular developeri, već ste se spajali na REST API-je iz frontenda. Sada ćete naučiti kako takav API izraditi na backend strani koristeći C#.

Struktura ASP.NET Core Web API projekta

Pomoću .NET CLI-ja ili Visual Studija možemo kreirati **templated** projekt Web API-ja. Primjer s CLI:

```
dotnet new webapi --use-controllers -o MojApiProjekt
```

```
cd MojApiProjekt
```

Ova naredba generirat će minimalni Web API projekt. Pregledajmo ključne dijelove strukture:

- **Program.cs** – ulazna točka aplikacije (umjesto Main metode kod konzolnih, ovdje konfiguriramo i pokrećemo web aplikaciju).
- **WeatherForecast.cs** – primjer *model* klase (generirana klasa s par svojstava, koristi se u demo svrhe).
- **Controllers/WeatherForecastController.cs** – primjer kontrolera (controller) koji ima nekolicinu ruta definiranih za GET metodu; Vraća nasumične podatke o vremenskoj prognozi.

Program.cs - konfiguracija aplikacije

Otvorite **Program.cs** iz generiranog projekta. Trebao bi se vidjeti kod koji izgleda slično:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

// Konfiguracija HTTP request pipelinea
if (app.Environment.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();
```

Budući da dolazite iz Angular svijeta, ovaj dio koda možda djeluje pomalo apstraktno, ali u suštini, ovdje se aplikacija priprema za rad.

Što se ovdje događa?

1. Kreiramo aplikaciju

```
var builder = WebApplication.CreateBuilder(args);
```

Ovo možemo usporediti s bootstrapanjem Angular aplikacije. Kreiramo objekt builder, koji sadrži sve što nam treba za konfiguraciju aplikacije – servise, konfiguraciju i middleware.

2. Dodajemo kontrolere

```
builder.Services.AddControllers();
```

Ova linija **registrira kontrolere**, slično kao što u Angularu definiramo rute (RouterModule.forRoot()). ASP.NET Core tada zna kako pronaći kontrolere i njihove rute.

3. Gradimo aplikaciju

```
var app = builder.Build();
```

Nakon što smo postavili servise, gradimo app objekt – nešto kao instanciranje glavne Angular aplikacije u main.ts.

4. Postavljamo pipeline (middleware)

Middleware su procesi koji se izvršavaju prilikom svakog HTTP zahtjeva, slično kao Angular interceptori. Sve dolje navedeno su middlewareovi sa svojom drugačijom svrhom.

```
if (app.Environment.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Ovo omogućuje **interaktivnu dokumentaciju API-ja** (Swagger UI), nešto poput Postmana, ali ugrađenog u browser.

Preusmjeravanje HTTP na HTTPS:

```
app.UseHttpsRedirection();
```

Slično kao kad u Angularu koristimo strict SSL pravila – ovaj **middleware** osigurava da se svi zahtjevi automatski preusmjere na HTTPS.

Autorizacija:

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

Middleware koji dodaje podršku za autentifikaciju/autorizaciju, npr. ako koristimo `[Authorize]` atributi na kontrolerima. Autentikacijski middleware ćemo kasnije podesiti.

Mapiranje kontrolera (rute):

```
app.MapControllers();
```

Middleware s kojim ASP.NET Core zna da treba slušati HTTP zahtjeve i proslijediti ih odgovarajućim kontrolerima, slično kao RouterModule u Angularu.

5. Pokrećemo aplikaciju

```
app.Run();
```

Ovo doslovno "pali" aplikaciju – pokreće ugrađeni **Kestrel web server**, koji će slušati zahtjeve na određenom portu.

 **Napomena:** Ovaj dio koda definira **kako se ASP.NET Core API pokreće i kako obrađuje HTTP zahtjeve**. Možete ga zamisliti kao main.ts u Angularu – postavljamo servise, konfiguraciju i inicijaliziramo aplikaciju.

Ne brinite ako vam još nije sve jasno – s vremenom će postati prirodnije kada krenemo s konkretnim API endpointima.

Kontroleri i rute

Sada dolazimo do **najvažnijeg dijela za nas – kontrolera**. Kontroleri su u ASP.NET Core Web API-ju ono što su **servisi (services)** i **rute (routing modules)** u Angular aplikaciji.

U mapi **Controllers** nalazi se **WeatherForecastController.cs**, koji je generiran po defaultu. Pogledajmo pojednostavljeni primjer jednog tipičnog API kontrolera:

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;

namespace MojApiProjekt.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class HelloController : ControllerBase {

        // GET: api/hello
        [HttpGet]
        public string Get() {
            return "Pozdrav iz Web API-ja!";
        }

        // GET: api/hello/ime
        [HttpGet("{name}")]
        public string GetPozdravByName(string name) {
            return $"Bok, {name}!";
        }
    }
}
```

Što ovdje radimo?

Definiramo API kontroler

[ApiController]

Ovaj atribut označava da je klasa API kontroler. To omogućuje neke korisne stvari, kao što su:

- Automatska validacija inputa
- Automatski model binding (preslikavanje podataka iz HTTP zahtjeva u metode)

Postavljamo rutu kontrolera

```
[Route("api/[controller]")]
```

Ovo postavlja **osnovnu rutu** za sve metode u kontroleru.

- [controller] će biti zamijenjeno imenom kontrolera, **bez sufiksa "Controller"**.
- Dakle, za **HelloController**, ruta će biti **api/hello**.
- To je slično Angular RouterModule.forRoot() gdje definiramo osnovne rute.

Nasleđujemo ControllerBase

```
public class HelloController : ControllerBase
```

ControllerBase osigurava korisne metode kao što su `Ok()`, `NotFound()`, `BadRequest()`, koje pomažu u vraćanju HTTP odgovora.

Možete ga zamisliti kao **Angular servis (@Injectable()) koji vraća HTTP odgovore**.

Metode u kontroleru – HTTP rute

1. GET zahtjev bez parametara

```
[HttpGet]
public string Get() {
    return "Pozdrav iz Web API-ja!";
}
```

Ova metoda odgovara na **GET zahtjeve na ruti api/hello**.

Vraća običan **string**, a ASP.NET Core automatski vraća **HTTP 200 OK** s tekstom.

Slično je Angular servisu (`HttpClient.get()`), ali na backend strani.

 **Primjer poziva iz browsera:**

 GET `http://localhost:5000/api/hello`

 **Odgovor:** "Pozdrav iz Web API-ja!"

2. GET zahtjev s parametrom

```
[HttpGet("{name}")]
public string GetPozdravByName(string name) {
    return $"Bok, {name}!";
}
```

Ova metoda prima **parametar iz URL-a** ({name}).

ASP.NET Core **automatski preslikava** {name} u argument metode.

Vraća **string s personaliziranim porukom**.



Primjer poziva iz browsera:

👉 GET http://localhost:5000/api/hello/Marko

◆ **Odgovor:** "Bok, Marko!"

To je slično **Angular ruti s parametrima**, npr. this.route.snapshot.paramMap.get('id') u Angular komponentama.

Vraćanje objekata umjesto plain stringova

U prethodnom primjeru, radi jednostavnosti koda metode u kontroleru su vraćale obične **stringove**. Međutim, u stvarnim API-jima obično želimo vraćati **objekte**, pogotovo kada šaljemo podatke frontend aplikaciji.

Evo poboljšane verzije GET metode koja vraća anonimni objekt:

```
// GET: api/hello/object
[HttpGet("object")]
public IActionResult GetObjectExample() {

    var result = new {
        Message = "Ovo će biti JSON",
        Time = DateTime.Now
    };

    return Ok(result);
}
```

Što se ovdje događa?

1. Vraćamo objekt umjesto stringa

Umjesto plain text odgovora, kreiramo **anonimni objekt** s dvije vrijednosti:

```
var result = new {
    Message = "Ovo će biti JSON",
    Time = DateTime.Now
};
```

Ovaj objekt će se **automatski pretvoriti u JSON** kada se vrati kao HTTP odgovor.

Osim anonimnog objekta kao u primjeru, možemo (i preporučeno je) vratiti tipizirani objekt, npr. **ExampleObjectResponse**, kojeg bi tada stvorili, postaviti mu 2 propertya (**Message** i **Time**) i stavili ga u primjerice “Models” folder. U oba slučaja rezultat će biti isti.

2. Koristimo **IActionResult** i **Ok(result)**

Umjesto string, metoda sada vraća **IActionResult**, što omogućuje fleksibilnije HTTP odgovore.

Ok(result) generira **HTTP 200 OK** status i vraća JSON objekt.

3. Automatska JSON serijalizacija

ASP.NET Core u pozadini koristi **System.Text.Json** za pretvaranje objekata u JSON.

Ne moramo raditi `JSON.stringify()` kao u JavaScriptu – backend to rješava automatski.

Kako izgleda odgovor?

Kada pozovemo GET `http://localhost:5000/api/hello/object`, dobit ćemo:

```
{
  "message": "Ovo će biti JSON",
  "time": "2025-03-15T13:34:15.12345+01:00"
}
```

To znači da se API ponaša **kao JSON servis**, slično kao kada u Angular aplikaciji koristimo `HttpClient.get<MyObject>()`.

Zaključak

Ključne stvari koje treba zapamtitи

- ✓ Kontroleri su srce API-ja – rješavaju HTTP zahtjeve i vraćaju odgovore
- ✓ Atribut `[Route("api/[controller]")]` određuje osnovnu rutu na temelju imena kontrolera.
- ✓ Metode koriste `[HttpGet]`, `[HttpPost]`, `[HttpPut]`, `[HttpDelete]` za definiranje HTTP zahtjeva.
- ✓ ASP.NET Core automatski preslikava URL parametre u metode, slično kao Angular rute.

- ✓ Umjesto stringova, vraćamo objekte koji se automatski serializiraju u JSON.
- ✓ Koristimo `IActionResult` za fleksibilnije HTTP odgovore.
- ✓ ASP.NET Core automatski pretvara objekte u JSON, bez potrebe za ručnim `JSON.stringify()`.
- ✓ Frontend aplikacija može jednostavno dohvatiti ove podatke koristeći `HttpClient.get()` u Angularu.

Pokretanje i testiranje API-ja

Sad kad imamo osnovni API postavljen, vrijeme je da ga pokrenemo i testiramo kako radi.

Pokretanje aplikacije

Ako koristite **VS Code**, otvorite terminal i pokrenite:

```
dotnet run
```

Ako koristite **Visual Studio**, samo kliknite na **Run** (zelena strelica), u tom slučaju otvoriti će se preglednik.

Što će se dogoditi?

U konzoli (Visual Studio također i automatski pokreće preglednik) će se pojaviti output koji prikazuje na kojem URL-u je aplikacija pokrenuta, obično:

- **http://localhost:5000** (za HTTP)
- **https://localhost:5001** (za HTTPS)

Testiranje API-ja u pregledniku

Otvorite preglednik i posjetite sljedeće URL-ove:

✓ Jednostavan GET endpoint:

👉 <http://localhost:5000/api/hello>

- ◆ Odgovor: "Pozdrav iz Web API-ja!"

✓ GET s parametrom:

👉 <http://localhost:5000/api/hello/Marko>

- ◆ Odgovor: "Bok, Marko!"

Testiranje API-ja preko Swaggera

Za složenije testiranje API-ja možeš koristiti:

- **Postman** (popularni alat za API testiranje)
- **Thunder Client** (VS Code ekstenzija)
- **Swagger UI** (ugrađeni alat u ASP.NET Core koji nam daje UI za testiranje API-ja)

Kako bi imali podršku za swagger potrebno je dodati paket u projekt navedenom naredbom (ili preko nuget managera u Visual Studiu)

```
dotnet add package Swashbuckle.AspNetCore
```

Tada ćemo unijeti slijedeće promjene u Program.cs:

```
Poslije builder.Services.AddControllers();  
builder.Services.AddSwaggerGen();
```

```
Prije app.UseHttpsRedirection();  
app.UseSwagger();  
app.UseSwaggerUI();
```

I to je to! Imati ćemo podršku za Swagger unutar naše aplikacije

✓ **Otvorite Swagger u pregledniku:**

👉 <http://localhost:5000/swagger>

◆ **Što će se prikazati?** Sučelje koje popisuje sve API rute i omogućuje **slanje testnih zahtjeva direktno iz preglednika.**

Životni ciklus HTTP zahtjeva u ASP.NET Core Web API

Kad klijent, poput **Angular aplikacije** ili preglednika, pošalje HTTP zahtjev, događa se niz koraka koji omogućuju njegovu obradu i vraćanje odgovora. Ovo nije nužno zapamtiti, ali pomaže razumjeti kako API funkcionira iznutra.

1. Zahtjev stiže na server

Klijent šalje HTTP zahtjev (npr. GET /api/hello/Marko) koji dolazi do **Kestrel web servera** – ugrađenog web servera u ASP.NET Core.

2. Middleware pipeline obrađuje zahtjev

Zahtjev prolazi kroz niz **middleware komponenti** – modula koji mogu presresti, modificirati ili odbaciti zahtjev prije nego što stigne do kontrolera.

Primjeri middleware-a u našem kodu:

- UseHttpsRedirection() – preusmjerava HTTP na HTTPS
- UseAuthorization() – provjerava ima li korisnik pristup
- UseSwaggerUI() – omogućuje Swagger sučelje

3. Routing određuje odgovarajući kontroler i metodu

ASP.NET Core Routing analizira URL i HTTP metodu kako bi odabralo odgovarajući kontroler i akciju. Primjer:

Klijent šalje GET /api/hello/Marko

Routing pronalazi **HelloController**

Poziva se metoda GetPozdravByName(string name), s **name = "Marko"**

4. Izvršava se metoda kontrolera

ASP.NET Core stvara instancu kontrolera (ako već ne postoji) pomoću dependency injectiona (DI).

Poziva se metoda koja odgovara ruti, npr.:

```
public string GetPozdravByName(string name) {  
    return $"Bok, {name}!";  
}
```

5. ASP.NET Core generira HTTP odgovor

ASP.NET Core automatski pretvara podatke u JSON ako je kontroler označen kao [ApiController]. Kreirani objekt:

```
return new { message = "Bok, Marko!" };
```

postaje HTTP odgovor

```
{ "message": "Bok, Marko!" }
```

6. HTTP odgovor se šalje klijentu

Klijent (npr. Angular aplikacija) prima HTTP odgovor koji sadrži **statusni kod i tijelo odgovora**.

Statusni kod može biti primjerice 200 (OK), 404 (NotFound), i sl.
Tijelo odgovora je prikazani JSON



Kontroleri su srce Web API-ja – svaki zahtjev prolazi kroz njih.

Middleware pipeline obrađuje zahtjev prije nego što stigne do kontrolera, slično Angular interceptorima.

Routing određuje koji kontroler i metoda će se pozvati, slično Angular RouterModule-u.

ASP.NET Core automatski pretvara objekte u JSON, baš kao što Angular HttpClient očekuje.

HTTP odgovor sadrži statusni kod i podatke koje frontend može prikazati korisniku.

Zaključak poglavlja

U ovom poglavlju upoznali smo se s osnovama **.NET Core** i **ASP.NET Core**, te vidjeli kako izgleda jedan **Web API projekt**. Ključna stvar koju treba razumjeti je **uloga kontrolera** i kako se **HTTP zahtjevi mapiraju na metode kontrolera**.

Ako ti se neki koncepti čine apstraktnima, ne brini – kroz praktične primjere sve će sjesti na svoje mjesto. Najbolji način za učenje je **pisanje stvarnog koda**.

Ako želite bolje razumjeti kako ASP.NET Core Web API funkcioniра ispod površine, preporučujem da samostalno istražite sljedeće teme:

- **Middleware u ASP.NET Core-u** – Middleware komponente obrađuju dolazne HTTP zahtjeve i odgovore prije nego što stignu do kontrolera. Istražite kako funkcioniра **Request Pipeline** i kako se dodavaju vlastite middleware komponente.
- **Životni ciklus HTTP zahtjeva** – Kako ASP.NET Core obraduje zahtjev od trenutka kad stigne do aplikacije, kako se preusmjerava kroz middleware i kako na kraju završava u kontroleru.
- **Detaljnije o kontrolerima** – lako smo vidjeli osnovnu strukturu kontrolera, istražite kako se koriste **[HttpGet]**, **[HttpPost]**, **[HttpPut]**, **[HttpDelete]** atributi te kako se može upravljati **parametrima u rutu i query stringovima**.
- **Model binding i validacija** – Kako ASP.NET Core pretvara dolazne podatke iz HTTP zahtjeva u objekte koje možeš koristiti unutar kontrolera.

Sada je vrijeme da "zaprljamo ruke"! U sljedećem poglavlju ćemo:

- ✓ Izraditi **vlastiti Web API endpoint**
- ✓ Implementirati jednostavne **CRUD operacije u memoriji**
- ✓ Učvrstiti znanje o **kontrolerima i rutama**

Tjedan 4: Izrada prvog Web API endpointa (CRUD u memoriji)

Cilj ovog poglavlja je izraditi praktičan Web API kontroler s više endpointa koji omogućuju osnovne CRUD operacije (Create, Read, Update, Delete) nad jednostavnim podatkovnim modelom. Koristit ćemo pohranu u memoriji (listu) umjesto prave baze podataka, kako bismo se usredotočili na mehaniku Web API-ja: definiranje ruta, rukovanje HTTP metodama i vraćanje odgovora.

U ovom poglavlju izgradit ćemo **Products API** – jednostavan API za upravljanje proizvodima, koristeći klasu Product koju smo već definirali.

Definiranje kontrolera i rute

Kreirajmo novi kontroler ProductsController. U ASP.NET Core kontekstu, dovoljno je dodati novu klasu u folder Controllers, koja ima sufiks "**Controller**" i potrebne **[ApiController]** i **[Route]** attribute. Kôd bi izgledao ovako:

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;

namespace MojApiProjekt.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        // Fiktivna "baza": statička lista proizvoda u memoriji
        private static List<Product> _products = new List<Product>()
        {
            new Product(1, "Tipkovnica", 45m),
            new Product(2, "Monitor", 330m),
            new Product(3, "Miš", 60m)
        };

        // GET: api/products
        [HttpGet]
        public ActionResult<IEnumerable<Product>> GetAll() {
            // Vraćamo cijelu listu proizvoda
            return Ok(_products);
        }

        // GET: api/products/2
        [HttpGet("{id}")]
        public ActionResult<Product> GetById(int id) {
            var product = _products.SingleOrDefault(p => p.Id == id);
            if (product == null) {
                return NotFound(); // vraća 404 ako proizvod nije nađen
            }
            return Ok(product);
        }

        // POST: api/products
        [HttpPost]
        public ActionResult<Product> Create(Product newProduct) {
            // Generiramo novi ID (npr. max id + 1)
            if (_products.Any())
                newProduct.Id = _products.Max(p => p.Id) + 1;
            else
                newProduct.Id = 1;

            _products.Add(newProduct);
        }
    }
}
```

```

        // Vraćamo 201 Created odgovor, sa lokacijom novog resursa
        return CreatedAtAction(nameof(GetById), new { id = newProduct.Id },
newProduct);
    }

    // PUT: api/products/2
    [HttpPut("{id}")]
    public IActionResult Update(int id, Product updatedProduct) {
        var product = _products.SingleOrDefault(p => p.Id == id);
        if (product == null) {
            return NotFound();
        }
        // Ažuriramo svojstva postojećeg proizvoda
        product.Name = updatedProduct.Name;
        product.Price = updatedProduct.Price;
        return NoContent(); // 204 No Content, uspješno bez sadržaja
    }

    // DELETE: api/products/3
    [HttpDelete("{id}")]
    public IActionResult Delete(int id) {
        var product = _products.SingleOrDefault(p => p.Id == id);
        if (product == null) {
            return NotFound();
        }
        _products.Remove(product);
        return NoContent();
    }
}
}

```

Ovaj kod omogućuje osnovne operacije nad kolekcijom proizvoda. Umjesto baze podataka koristimo **statičku listu** kako bismo se fokusirali na API logiku.

Puno toga se događa u ovom isječku, pa razložimo korak po korak:

- **_products:** je lista proizvoda koja simulira bazu podataka. Početno smo dodali nekoliko proizvoda (ID 1, 2, 3).
- **GetAll():** odgovara na **GET /api/products** i vraća popis svih proizvoda. Povratni tip **IActionResult<IEnumerable<Product>>** omogućuje vraćanje i podataka i statusnog koda. Vraćamo **Ok(_products)**, što klijent prima kao **HTTP 200 OK** i JSON listu

- **GetById(int id)**: odgovara na `GET /api/products/2`. ASP.NET Core automatski prepoznaje broj iz URL-a i proslijeđuje ga metodi. Ako proizvod postoji, vraća ga u JSON formatu (`Ok(product)`). Ako ne postoji, vraća **404 Not Found**.
- **Create(Product newProduct)**: omogućuje dodavanje proizvoda putem `POST /api/products`. ASP.NET Core automatski pretvara JSON iz zahtjeva u Product objekt. Generiramo novi ID i dodajemo proizvod u listu. Koristimo `CreatedAtAction()`, što vraća **201 Created** i lokaciju novog resursa.
- **Update(int id, Product updatedProduct)**: omogućuje ažuriranje putem `PUT /api/products/2`. Ako proizvod postoji, mijenjamo njegova svojstva i vraćamo **204 No Content**. Ako ne postoji, vraćamo **404 Not Found**.
- **Delete(int id)**: briše proizvod putem `DELETE /api/products/2`. Ako proizvod postoji, uklanjamo ga iz liste i vraćamo **204 No Content**, inače vraćamo **404 Not Found**.

Ovim smo pokrili svih pet glavnih RESTful operacija za kolekciju resursa "products":

- ✓ **GET** svi proizvodi
- ✓ **GET** jedan proizvod po ID-u
- ✓ **POST** novi proizvod
- ✓ **PUT** ažuriranje proizvoda
- ✓ **DELETE** brisanje proizvoda

 U praksi, podaci bi se dohvaćali i spremali u bazu podataka, ali za sada koristimo memoriju kako bismo se fokusirali na rad s kontrolerima i API metodama.

Ispitivanje CRUD endpointa

Pokrenimo aplikaciju i koristimo alat po izboru (Swagger UI, Postman, curl) kako bi testirali:

- GET /api/products → trebalo bi dobiti listu od 3 proizvoda koje smo inicijalno postavili.
- GET /api/products/2 → trebalo bi dobiti proizvod s id=2 (Monitor).
- GET /api/products/999 → trebalo bi dobiti 404 Not Found (prazno tijelo).
- POST /api/products sa JSON tijelom npr. { "name": "Stolica", "price": 100 } → trebao bi dobiti 201 Created, a u tijelu novokreirani objekt, recimo { id: 4, name: "Stolica", price: 100 }. Ako pogledamo header Location, bit će /api/products/4.
- PUT /api/products/4 s JSON tijelom { "id": 4, "name": "Uredska stolica", "price": 150 } → dobit će se 204 No Content. Ako nakon toga GET-om se dohvati /api/products/4, naziv će biti promijenjen.
- DELETE /api/products/4 → vraća 204 No Content. Nakon toga GET /api/products/4 trebao bi vratiti 404.

Ovo je prilično dobro, zar ne? Uz relativno malo koda dobili smo funkcionalan API sa svih 5 glavnih RESTful operacija.

 **Napomena:** Ovdje nema trajnog spremanja - ako restartamo aplikaciju, lista će se vratiti na početne tri stavke, jer podatke držimo u memoriji. Rješavanje *perzistencije* riješit ćemo u idućem poglavlju integracijom baze podataka.

JSON serijalizacija i povratne vrijednosti

Kada vratimo **objekt** ili **listu objekata** iz metode kontrolera, ASP.NET Core automatski ih **pretvara u JSON**. Zahvaljujući [ApiController] atributu i zadanoj konfiguraciji, ne moramo ručno raditi JsonConvert.SerializeObject().

- **Backend vraća C# objekt, a klijent dobiva JSON – bez dodatnog koda!**
- **Manje posla, brža implementacija.**

Praktični savjeti za vraćanje odgovora

Kada koristiti konkretan tip, a kada ActionResult<T>?

- **Ako metoda uvijek vraća uspjeh**, može se na primjer vratiti **IEnumerable<Product>** i [ApiController] će automatski postaviti status **200 OK**.
- **Ako može doći do greške (npr. ne postoji traženi proizvod)**, treba koristiti **ActionResult<Product>** ili samo **IActionResult**, pa ručno postavi odgovarajući status (Ok(), NotFound(), BadRequest()).

Koji statusni kod koristiti?

U REST API-ju, statusni kodovi su važni jer klijent pomoću njih zna što se dogodilo. Ovo su **najčešće korišteni**:

- 200 OK** – Sve je prošlo u redu (npr. uspješan GET).
- 201 Created** – Kada POST metoda kreira novi resurs.
- 204 No Content** – Kada PUT ili DELETE uspije, ali nema tijelo odgovora.
- 400 Bad Request** – Nešto nije u redu s podacima koje je poslao klijent.
- 404 Not Found** – Traženi resurs ne postoji.
- 401 Unauthorized** – Korisnik nije prijavljen.
- 403 Forbidden** – Korisnik nema pravo pristupa.

 **Držimo se REST konvencija** – npr. 201 Created za POST, 404 za "not found", 204 za brisanje bez sadržaja, itd. To pomaže frontend developerima da lako razumiju odgovore API-ja.

Zaključak poglavlja

Trenutno, naš API **radi**, ali podaci se čuvaju u običnoj listi u memoriji. To znači da se sve gubi kada ponovno pokrenemo aplikaciju. U pravoj aplikaciji želimo da podaci **ostanu sačuvani**, što znači da je vrijeme za bazu podataka!

Možete dodatno proširiti svoje znanje istražujući sljedeće:

- **Naprednije rute i parametre** – Kako proslijediti putem URL-a, query stringa ili tijela zahtjeva? Kako koristiti atributi poput `[FromQuery]`, `[FromRoute]` i `[FromBody]`?
- **Statusni kodovi HTTP odgovora** – Kada koristiti **200 OK**, **201 Created**, **400 Bad Request**, **404 Not Found**? Kako ih eksplisitno vratiti unutar metoda?
- **Kako ubaciti ILogger** u konstruktor kontrolera i logirati informaciju u konzoli?

U sljedećim poglavljima naučit ćemo kako:

- ✓ **Spojiti bazu podataka** pomoću **Entity Framework Core ORM-a** i zamijeniti memorisku listu pravom tablicom.
- ✓ **Razumjeti Dependency Injection (DI)** – koncept koji već koristimo, ali ga do sada nismo eksplisitno objašnjavali.

Što je Dependency Injection i zašto je bitan?

U ASP.NET Core aplikaciji, kontroleri se ne instanciraju ručno – framework ih **automatski kreira** i ubacuje potrebne komponente. Ovo već koristimo nesvjesno, npr. kada u kontroleru dodamo `ILogger<T>` ili `IConfiguration` u konstruktor, ASP.NET ih sam ubaci.

Prije nego krenemo s bazom, u sljedećem poglavlju ćemo detaljnije objasniti **Dependency Injection**, kako radi i kako ga iskoristiti za bolju organizaciju koda. Nakon toga, implementirat ćemo **vlastiti servis** koji će komunicirati s bazom.

Tjedan 5: Dependency Injection u .NET Core-u

Cilj ovog poglavlja je razumjeti koncept Dependency Injection-a (DI) i način na koji ga koristi ASP.NET Core. Naučit ćemo kako registrirati i koristiti vlastite servise putem DI mehanizma unutar Web API-ja. Ovaj pristup ima sličnosti s konceptom servisa i injektiranja u Angularu, pa ćemo iskoristiti isto razmišljanje na backendu.

Što je Dependency Injection?

Dependency Injection (DI) je *design pattern* koji omogućuje **slabu povezanost** (loose coupling) između komponenti. Umjesto da klasa sama kreira instance svojih ovisnosti (`[new SomeService()]` unutar kontrolera), DI omogućuje da **ovisnosti dobijemo izvana**, najčešće putem konstruktora.

Kako to funkcionira u ASP.NET Core?

- ASP.NET Core ima **ugrađeni Dependency Injection**.
- Postoji **centralni IoC (Inversion of Control) container** gdje registriramo servise i objekte.
- Kada ASP.NET Core kreira **kontroler**, on **automatski ubacuje sve potrebne ovisnosti**.

Kako je to slično Angularu?

Ako se radi s Angularom, ovo je slično:

- U Angularu, servis označimo s **@Injectable** i Angularov injector nam ga automatski ubaci u konstruktor komponente.
- U ASP.NET Core, **DI container** automatski ubacuje servis u konstruktor kontrolera – na isti način kao Angular injector.

Jednostavno rečeno: ASP.NET Core sam rješava ovisnosti umjesto nas!

Registracija servisa u ASP.NET Core

Registracija se radi u dijelu koda `builder.Services.Add...` u **Program.cs** datoteci. Već smo vidjeli `builder.Services.AddControllers()`. To je ekstenzijska metoda koja unutra registrira sve kontrolere i potrebne servise za njih. Mi možemo registrirati i vlastite tipove.

Sintaksa je obično slična:

```
builder.Services.AddTransient<IMojServis, MojServis>();
```

Možemo koristiti i **AddScoped** ili **AddSingleton**, ovisno o **lifetimeu servisa** (koliko dugo instanca živi). U Web API-ju, **Scoped** se najčešće koristi za servise koji ovise o kontekstu HTTP zahtjeva, poput **DB konteksta**.

Detaljnije objašnjenje slijedi kasnije.

Kreirajmo jednostavan servis i injektirajmo ga

Primjer: Napraviti ćemo servis koji daje trenutno vrijeme u string formatu (pretpostavimo da nam to treba na više mesta, pa ne želimo duplicirati kod `DateTime.Now` formatiranje unutar kontrolera, već za to imamo servis `TimeService`).

1. Definiramo sučelje (interface) i implementaciju servisa:

```
public interface ITimeService {
    string GetcurrentTime();
}

public class TimeService : ITimeService {
    public string GetcurrentTime() {

        // Vraća trenutno vrijeme kao string
        return DateTime.Now.ToString("HH:mm:ss");
    }
}
```

Ovdje je `ITimeService` sučelje koje deklariše metodu, a `TimeService` konkretna implementacija. Dobra praksa u OOP je injektirati sučelja, ne konkretnе klase (radi fleksibilnosti i testiranja), pa ćemo to i slijediti.

2. Registriramo servis u Program.cs:

U `Program.cs`, prije `builder.Build()`, treba dodati:

```
builder.Services.AddScoped<ITimeService, TimeService>();
```

Koristimo **AddScoped** jer želimo da servis traje samo tijekom trajanja jednog *requesta*. Lako je u ovom primjeru servis jednostavan, u stvarnim scenarijima može čuvati podatke specifične za taj zahtjev.

Što to znači u praksi?

- Svaki put kada neki dio aplikacije (npr. kontroler) zatraži **ITimeService**, DI container će mu dati **novu instancu** servisa **TimeService**, ali samo za taj zahtjev.
- Ako se isti servis zatraži **negdje dalje unutar istog HTTP requesta**, koristit će se **ista instanca**.
- Čim zahtjev završi, instanca se odbacuje i više se ne koristi.

3. Korištenje servisa u kontroleru:

Napravimo, primjerice, **HomeController**, ili možemo koristiti postojeći **ProductsController**. Ipak, bolje je imati zaseban kontroler za ovaj demo kako bismo ga jasno odvojili i olakšali testiranje:

```
[ApiController]
[Route("api/[controller]")]
public class HomeController : ControllerBase {
    private readonly ITimeService _timeService;

    // Kroz konstruktor tražimo ITimeService
    public HomeController(ITimeService timeService) {
        _timeService = timeService;
    }

    // GET: api/home/time
    [HttpGet("time")]
    public string GetTime() {
        return $"Trenutno vrijeme: {_timeService.GetCurrentTime()}";
    }
}
```

Kada netko pozove **GET /api/home/time**, ASP.NET Core će napraviti sljedeće korake:

1. **Instancira HomeController**, jer je to kontroler koji obrađuje zahtjev.
2. **Provjeri konstruktor HomeController** i vidi da mu je potreban ITimeService.
3. **Pogleda u DI container** i pronade registraciju za ITimeService koja upućuje na TimeService.
 - o Budući da je registriran s **scoped** lifetimeom, kreira novu instancu TimeService za taj zahtjev.
4. **Ubaci tu instancu u kontroler** i izvrši metodu GetTime().
5. **Vrati rezultat metode** klijentu – ASP.NET Core će automatski pretvoriti odgovor u JSON string ili običan tekst, ovisno o vrsti podatka.

Ako se pozove taj endpoint, dobit će se JSON string s vremenom, npr:

"Trenutno vrijeme: 14:30:05"

 Iako vraćamo string, ASP.NET zbog ApiController atributa pretvara odgovor kao JSON string, što je za očekivati.

Zašto je DI koristan?

Većina stvari u ASP.NET Core radi preko DI:

- **DbContext** (o tome uskoro) se injektira u kontroler.
- **Konfiguracija** (IConfiguration) se može injektirati.
- **Loggeri** (ILogger<T>) se injektiraju.
- **Poslovne komponente** ili servisi se injektiraju jedni u druge umjesto da imaju konstruktore razbacane na raznim dijelovima aplikacije.

Prednosti:

- **Lakše testiranje:** moguće je primjerice napraviti *fake implementaciju* ITimeService za potrebe unit testova i injektirati je u kontroler umjesto prave, što omogućuje izolirano testiranje kontrolera.
- **Modularnost:** Kod je čišći, kontroler se ne bavi instanciranjem, samo deklarira što mu treba.
- **Životni vijek objekata je kontroliran:** npr. DB kontekst je po requestu, što sprječava memory leakove i konflikte između zahtjeva.

Za developer, trebamo samo zapamtiti:

- Registriraju se sve potrebne servise u Program.cs (AddScoped/Transient/Singleton).
- U konstruktoru bilo koje klase koju stvara framework (kontroler, pa i drugi servisi ako se jedan servis ubacuje u drugi) navede se što treba i dobit će se instancirano.

Napomena o lifetime: Transient vs Scoped vs Singleton

ASP.NET Core nudi tri načina registracije servisa u **Dependency Injection (DI) containeru**:

- **AddTransient** – Stvara **novu instancu svaki put** kad se servis zatraži.
 - Koristi se za **stateless** servise koji su **jeftini za kreiranje** i ne zadržavaju stanje.
 - Primjer: **lagani helper servisi**, poput servisa za formatiranje podataka.
- **AddScoped** – Stvara **jednu instancu po HTTP zahtjevu**.
 - Unutar istog requesta svi dijelovi aplikacije koriste **istu instancu**, ali kod novog zahtjeva se kreira nova.
 - Najčešće se koristi za **servise koji rade s bazom podataka**, jer svaki request dobiva svoju instancu.
- **AddSingleton** – Stvara **jednu instancu** koja traje **cijeli vijek trajanja aplikacije**.
 - Dobar za **caching servise** ili objekte koje želiš dijeliti među svim korisnicima.
 - **Nije preporučeno za DB kontekst** ili objekte koji nisu **thread-safe**, jer može dovesti do problema s konkurentnim pristupom.

Pravilno biranje **lifetimea** ključno je za **performanse i stabilnost** aplikacije.

Zaključak poglavlja

Sada smo u naš projekt dodali **vlastiti servis** koristeći **Dependency Injection (DI) container**. Možda se čini kao dodatni korak za nešto jednostavno poput dohvaćanja vremena, ali važno je razumjeti koncept, jer će se DI koristiti u svim složenijim dijelovima aplikacije.

U idućim poglavlјima **DI će se podrazumijevati** – primjerice, **EF Core DbContext** ćemo registrirati i injektirati u kontroler. Također, ako aplikaciju proširimo, možda ćemo uvesti **servisni sloj** između kontrolera i baze za dodatnu logiku.

Ako želite bolje razumjeti DI u .NET Core-u, možete istražiti sljedeće:

- Životni vijek servisa (Transient, Scoped, Singleton) – kada koristiti koji i kakav utjecaj imaju na aplikaciju.
- Kako DI pojednostavljuje testiranje – ovisnosti se mogu lako zamijeniti mock objektima.
- Factory pattern u DI-ju – kako dinamički kreirati servise umjesto da ih uvijek registriramo unaprijed.
- Opcije za registraciju servisa – registracija putem interfejsa, registracija konkretnih klasa.
- Kako DI funkcioniра ispod haube – istražite kako ASP.NET Core automatski upravlja instanciranjem objekata.

Ključna stvar za zapamtitи:

ASP.NET Core ima ugrađen Dependency Injection i koristi ga za sve kontrolere ([ApiController] kontroleri su automatski kreirani putem DI-ja).

Sljedeće poglavlje bavit će se **integracijom baze podataka putem Entity Framework Core**. Naučit ćemo kako koristiti **DI i prethodno naučene koncepte** kako bismo zamjenili **in-memory listu** pravom bazom podataka.

Tjedan 6: Integracija baze podataka s Entity Framework Core

Cilj ovog poglavlja je povezati naš Web API s bazom podataka koristeći Entity Framework Core (EF Core), popularni .NET ORM (Object-Relational Mapper). Naučit ćemo kako kreirati model baze pomoću Code-First pristupa, konfigurirati kontekst baze (DbContext) i izvoditi osnovne CRUD operacije izravno nad bazom umjesto in-memory liste. Također ćemo ukratko obraditi migracije baze kako bismo iz koda generirali tablice.

Uvod u Entity Framework Core

Entity Framework Core (EF Core) omogućuje rad s bazom podataka koristeći **C# klase**, bez potrebe za ručnim pisanjem SQL upita za osnovne operacije.

Kako to funkcionira?

- Definiramo **entitete** (C# klase) koji predstavljaju **tablice u bazi**.
- Kreiramo **DbContext klasu**, koja služi kao **veza s bazom podataka** i sadrži kolekcije (**DbSet<T>**) za entitete.
- Kada koristimo **DbSet<T>**, EF Core automatski **prevodi naše C# operacije u SQL upite** za bazu podataka.
- Možemo koristiti različite baze (SQL Server, SQLite, PostgreSQL itd.) jednostavno mijenjajući **provider**.

Koju bazu koristimo u primjeru?

Za naš primjer koristit ćemo **SQLite** – jednostavnu **datotečnu bazu** koja ne zahtijeva server, što je idealno za razvoj i testiranje.

Nastavit ćemo raditi s **Product klasom**, koja će sada postati **pravi entitet u bazi**.

Postavljanje EF Core u projekt

Prije pisanja koda, trebamo osigurati da su potrebni EF Core paketi instalirani:

- Osnovni Microsoft.EntityFrameworkCore (automatski se dodaje s **providerom**)
- Provider za željenu bazu. Npr. za SQLite: **Microsoft.EntityFrameworkCore.Sqlite**. Za SQL Server: **Microsoft.EntityFrameworkCore.SqlServer**.

Ako se koristi dotnet CLI:

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

(ili u Visual Studio preko NuGet upravitelja dodati paket).

Također za migracije (o tome uskoro) trebamo koristiti paket i pomoćni alat

- **Microsoft.EntityFrameworkCore.Tools**
- **dotnet-ef**

pa ćemo i njih dodati, također naredbom

```
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

```
dotnet tool install --global dotnet-ef
```

Definiranje DbContext-a i modela

Već imamo klasu **Product**, koja će predstavljati tablicu **Products** u bazi. Sada trebamo definirati **DbContext**, koji će upravljati vezom s bazom i omogućiti nam rad s podacima:

```
using Microsoft.EntityFrameworkCore;

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

    // DbSet predstavlja tablicu u bazi za entitet Product
    public DbSet<Product> Products { get; set; }
}
```

Što se ovdje događa?

- **AppDbContext nasljeđuje DbContext**
 - DbContext je osnovna klasa u EF Core koja omogućuje komunikaciju s bazom.
- **Konstruktor prima DbContextOptions<AppDbContext>**
 - Ovaj objekt sadrži **konfiguraciju baze** (npr. connection string, provider).
 - Prosljeđujemo ga bazi koristeći base(options), što omogućuje EF Core-u da zna koju bazu koristiti.
- **DbSet<Product> Products**
 - DbSet predstavlja tablicu u bazi.
 - Po defaultu, EF Core će tablicu nazvati "**Products**" (ime propertyja u množini postaje naziv tablice).

Sada kada imamo **DbContext**, sljedeći korak je **povezivanje s aplikacijom** i registracija u DI containeru.

Registracija DbContext-a u DI i konfiguracija konekcije

U **Program.cs**, trebamo registrirati AppDbContext kao servis i reći EF Core-u koju bazu i connection string da koristi.

Primjer, ako se držimo SQLite:

1. Dodati ćemo u **appsettings.json** connection string (ništa nas ne sprečava da ga navedemo u kodu, ali dobra je praksa koristiti konfiguraciju):

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=products.db"
  }
}
```

(Prepostavimo da imamo već appsettings.json u projektu; u poglavlju 7 ćemo detaljnije o konfiguraciji, ali možemo i ovdje na brzinu ga promjeniti).

2. U Program.cs dodati ćemo using za **Microsoft.EntityFrameworkCore** na vrhu, pa:

```
string connectionString =
builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<AppDbContext>(options =>
  options.UseSqlite(connectionString));
```

(također prije `builder.build()` poziva)

Ovdje koristimo UseSqlite ekstenziju (dolazi s paketom EFCore.Sqlite) da konfiguriramo SQLite provider i stavljamo connection string.

Ako se ne želi kroz konfiguraciju, moglo bi se i direktno upisati `options.UseSqlite("Data Source=products.db")`, ali bolje se naviknuti na config.

Time je AppDbContext dostupan kroz DI. EF Core će osigurati da je registriran kao Scoped po defaultu, što je dobro (jedan kontekst po requestu).

Ažuriranje kontrolera na EF Core

Vrijeme je da **ProductsController** prestane koristiti `_products` listu u memoriji i počne dohvaćati podatke iz baze.

Što se mijenja?

- Umjesto da podaci budu pohranjeni u običnoj listi, koristit ćemo **EF Core i SQLite bazu**.
- **AppDbContext** će se **injektirati putem konstruktora** i omogućiti nam rad s podacima kroz `_context.Products`.

Dobra vijest je da se kod neće previše zakomplicirati – API metode će i dalje raditi na isti način, ali sada s pravom bazom podataka.

Evo kako izgleda **izmijenjeni ProductsController s EF Core**:

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

    public ProductsController(AppDbContext context) {
        _context = context;
    }

    // GET: api/products
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Product>>> GetAll() {
        var products = await _context.Products.ToListAsync();
        return Ok(products);
    }

    // GET: api/products/{id}
    [HttpGet("{id}")]
    public async Task<ActionResult<Product>> GetById(int id) {
        var product = await _context.Products.FindAsync(id);
        if (product == null) {
            return NotFound();
        }
        return Ok(product);
    }

    // POST: api/products
    [HttpPost]
    public async Task<ActionResult<Product>> Create(Product newProduct) {
        _context.Products.Add(newProduct);
        await _context.SaveChangesAsync();

        // nakon SaveChangesAsync, newProduct.Id će biti popunjeno
        return CreatedAtAction(nameof(GetById), new { id = newProduct.Id },
newProduct);
    }
}
```

```
}

// PUT: api/products/{id}
[HttpPut("{id}")]
public async Task<IActionResult> Update(int id, Product updatedProduct) {
    if (id != updatedProduct.Id) {

        // ID iz URL-a i tijela se ne podudaraju, to je bad request
        return BadRequest();
    }

    var product = await _context.Products.FindAsync(id);

    if (product == null) {
        return NotFound();
    }

    // Ažuriramo postojeći entitet
    product.Name = updatedProduct.Name;
    product.Price = updatedProduct.Price;
    await _context.SaveChangesAsync();

    return NoContent();
}

// DELETE: api/products/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(int id) {
    var product = await _context.Products.FindAsync(id);
    if (product == null) {

        return NotFound();
    }

    _context.Products.Remove(product);
    await _context.SaveChangesAsync();

    return NoContent();
}
```

Što se promijenilo u kodu?

Sada **ProductsController** koristi **bazu podataka** umjesto liste u memoriji. Evo ključnih razlika i objašnjenja:

Konstruktor sada prima ApplicationDbContext i sprema ga u _context

- Umjesto da podaci budu u listi, sada ih dohvaćamo iz baze.

Sve metode su sada asinkrone (async Task<...>)

- **EF Core koristi async metode** (`ToListAsync`, `FindAsync`, `SaveChangesAsync`), kako bismo izbjegli blokiranje glavnog thread-a.
- Ovo funkcionira slično **JavaScript async/await sa Promise-ima** – omogućuje API-ju da ostane responzivan dok čeka bazu.

Dohvaćanje svih proizvoda (GetAll)

- Umjesto `_products`, sada koristimo `_context.Products.ToListAsync()`, što dohvaca podatke iz baze.

Dohvaćanje po ID-u (GetById)

- `FindAsync(id)` traži entitet po primarnom ključu (Id).
- Ako proizvod ne postoji, vraćamo `NotFound()`.

Kreiranje proizvoda (Create)

- **Nema više ručnog dodjeljivanja Id polja** – prepostavljamo da baza koristi **auto-increment ključ** (npr. SQLite automatski generira Id kada je definiran kao INTEGER PRIMARY KEY AUTOINCREMENT).
- Tek nakon poziva `_context.SaveChangesAsync()`, novi Id će biti stvarno dodijeljen objektu.

Ažuriranje proizvoda (Update)

- Dodali smo provjeru: **uspoređujemo id iz URL-a s Id iz tijela zahtjeva**.
- Ovo sprječava slučajne greške ako klijent pošalje neuskladene podatke.

Brisanje proizvoda (Delete)

- Logika je ista – tražimo proizvod, brišemo ga i spremamo promjene u bazu.

Sljedeći korak?

Dodali smo **DbContext** u naš projekt i promijenili endpoint-e da rade s tablicom, umjesto s običnom listom u memoriji. No, još nismo gotovi!

Trenutno, entiteti su definirani u kodu, ali baza još **fizički ne postoji** – moramo napraviti **migraciju baze**, koja će stvoriti tablice u SQLite bazi.

U sljedećem koraku:

- **Kreirat ćemo migraciju** kako bi Entity Framework Core generirao potrebne SQL naredbe za stvaranje baze.
- **Primijenit ćemo migraciju**, što će fizički stvoriti bazu i tablice.
- **Provjeriti radi li API sada s pravim podacima**, umjesto s podacima u memoriji.



S ovim kodom, naše operacije sada **stvarno utječu na bazu podataka**.

Migracije i kreiranje baze

EF Core podržava **code-first** pristup: definiramo modele i kontekst, a onda generiramo migracije koje kreiraju bazu. Evo kako:

1. U terminalu (unutar foldera projekta) dodajemo migraciju:

```
dotnet ef migrations add InitialCreate
```

Ovo će stvoriti folder **Migrations** s migracijskom klasom. (Ako dobijete grešku "ef not found", instalirajte paket `Microsoft.EntityFrameworkCore.Tools` i `dotnet-ef` alat kako je bilo navedeno prethodno)

2. Primijenimo migracije na bazu:

```
dotnet ef database update
```

Ovo će stvoriti datoteku **products.db** (jer smo tako naveli u connection stringu) i unutar nje tablicu **Products** s kolumnama Id, Name, Price. Također, EF će kreirati i tablicu `_EFMigrationsHistory` za praćenje migracija.

Testiranje s bazom

Ponovno pokrenimo aplikaciju. Sada ćemo testirati one iste endpoint pozive:

- GET /api/products → prazan popis (ako baza nema podataka, a vjerojatno je prazna nakon kreiranja).
- POST /api/products → dodajemo neki proizvod. Trebao bi upisati red u bazu.
- GET ponovno → sad vidimo taj proizvod.

Čestitam, napravljen je prvi integrirani Web API s bazom podataka!

Primijetite da se kôd kontrolera nije dramatično zakomplicirao prelaskom na bazu. EF Core čini pristup bazi vrlo sličnim radu s kolekcijom u memoriji.

Još malo o EF Core

Iako je ovo dovoljno za osnovne CRUD operacije, evo nekoliko važnih stvari koje trebate znati o Entity Framework Core-u:

Praćenje entiteta (Tracking)

Kada dohvate podatak iz baze, EF Core ga **prati** u memoriji.

Ako promijenite neko svojstvo (npr. `product.Name = "Novi naziv";`) i pozovete `_context.SaveChangesAsync();`, EF Core automatski prepozna promjenu i generira **SQL UPDATE**.

Ako ne želite da EF Core prati promjene (npr. ako samo čitate podatke, bez izmjena), koristite `.AsNoTracking()` da poboljšate performanse:

```
var products = await _context.Products.AsNoTracking().ToListAsync();
```

Brisanje podataka

Brisanje je vrlo jednostavno: dovoljno je pozvati `.Remove()` i zatim `.SaveChangesAsync()`.

EF Core automatski generira SQL DELETE upit:

```
_context.Products.Remove(product);
await _context.SaveChangesAsync();
```

Filtriranje podataka pomoću LINQ-a

Možete koristiti LINQ upite za filtriranje podataka, što je vrlo praktično.

Na primjer, dohvate sve proizvode skuplje od 100€:

```
var skupiProizvodi = await _context.Products
    .Where(p => p.Price > 100)
    .ToListAsync();
```

Što još može EF Core?

EF Core nudi **mnoge dodatne mogućnosti**, poput:

- Relacija između entiteta (npr. **one-to-many**, **many-to-many** odnosi).
- Ograničenja i validacije (npr. polja koja ne smiju biti prazna).
- Automatski generirana svojstva (npr. **datum kreiranja** ili **posljednje izmjene**).
- **Lazy Loading** i **Eager Loading** za efikasnije dohvaćanje podataka.

 Za početak, **jednostavna tablica i CRUD operacije su sasvim dovoljni**. Kasnije, kada budete spremni, možete istražiti naprednije značajke i dalje optimizirati svoj API

Zaključak poglavlja

Frontend sada komunicira s našim Web API-jem, a API je povezan s bazom podataka putem **Entity Framework Core-a**. Sada naš API zaista "pamti" podatke, što je ogroman korak prema pravoj aplikaciji.

U sljedećim poglavljima dodat ćemo nekoliko ključnih elemenata:

- Konfiguraciju** – kako upravljati **connection stringom** i drugim postavkama.
- Sigurnost** – osnove **autentifikacije i autorizacije**, kako bi API bio zaštićen.
- Finalni projekt** – sve ćemo povezati u **zaokruženu aplikaciju**.

Mali izazov: isprobajte kako EF Core upravlja promjenama u bazi.

Dodajte novo svojstvo (property) u Product klasu (npr. `Description`).

Pokrenite **add migration**:

```
dotnet ef migrations add AddDescription
```

Ažurirajte bazu s **database update**

```
dotnet ef database update
```

Pogledajte kako EF Core dodaje **novu klasu unutar Migrations foldera**

Sada kada znamo kako pohraniti podatke, vrijeme je da prijeđemo na sljedeću veliku temu:

Konfiguracija aplikacije

Tjedan 7: Konfiguracija aplikacije i appsettings.json

Cilj ovog poglavlja je razumjeti kako ASP.NET Core upravlja konfiguracijom aplikacije, s naglaskom na **appsettings.json** datoteku. Naučit ćemo kako čitati vrijednosti iz konfiguracije, poput connection stringa i prilagođenih postavki, te ih koristiti unutar aplikacije. To ćemo demonstrirati na primjeru dohvatanja connection stringa (koji smo već koristili) i jedne prilagođene postavke.

appsettings.json i konfiguracijski sustav

U **.NET Core** projektima koristi se **appsettings.json** kao glavna konfiguracijska datoteka. Kada kreiramo novi Web API projekt, **Visual Studio automatski generira dvije konfiguracijske datoteke**:

- **appsettings.json** – sadrži osnovne (defaultne) postavke.
- **appsettings.Development.json** – omogućuje preklapanje postavki specifičnih za razvojno okruženje, primjerice detaljnije logiranje.

Mi ćemo se fokusirati na **appsettings.json**, jer će on sadržati ključne informacije, poput **connection stringa**, postavki API-ja i drugih konfiguracija koje će nam trebati u nastavku.

Primjer sadržaja appsettings.json može biti:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=products.db"
  },
  "AppSettings": {
    "ApiName": "ProductsAPI",
    "Version": "1.0"
  }
}
```

Ovdje možemo primijetiti nekoliko ključnih sekcija:

- **Logging** i **AllowedHosts** – možemo ih zanemariti jer nisu u fokusu ovog priručnika.
- **ConnectionStrings** – ovdje se često definiraju konekcije na bazu podataka. U našem primjeru dodan je "**DefaultConnection**".
- **AppSettings** – prilagođena sekcija, mogla je imati bilo koje ime. Ovdje smo, primjerice, pohranili naziv API-ja i verziju.

ASP.NET Core **automatski učitava appsettings.json pri pokretanju aplikacije** i spaja ga s drugim izvorima konfiguracije, kao što su **environment varijable** ili **user secrets**. Međutim, za sada nam to nije važno. Sve konfiguracijske vrijednosti mogu se dohvatiti u kodu putem **IConfiguration** objekta.

Čitanje konfiguracije u kodu

Već smo u prošlom poglavlju vidjeli jedan način:

`builder.Configuration.GetConnectionString("DefaultConnection")` unutar Program.cs, što je najdirektniji način za connection string.

No, konfiguraciju možemo čitati i bilo gdje drugdje, npr. unutar kontrolera ili servisa, i to na dva načina:

1. **Direktno korištenje IConfiguration** objekta (koji je također registriran u DI).
2. Korištenje "Options pattern" – mapiranje sekcije na strongly-typed POCO klasu. (Spomenut ćemo, ali nećemo ulaziti duboko, jer to je malo više teorije nego što nam treba sada.)

1. Injektiranje IConfiguration

IConfiguration je dostupan kroz DI, što znači da ga možemo tražiti u konstruktoru kontrolera ili servisa. Napravimo primjer kontrolera koji koristi konfiguraciju:

```
[ApiController]
[Route("api/[controller]")]
public class InfoController : ControllerBase {

    private readonly IConfiguration _config;

    public InfoController(IConfiguration config) {
        _config = config;
    }

    // GET: api/info/version
    [HttpGet("version")]
    public ActionResult<string> GetVersion() {
        string apiName = _config["AppSettings:ApiName"];
        string version = _config["AppSettings:Version"];
        return Ok($"{apiName} - Verzija {version}");
    }
}
```

Objasnilo:

- Injektirali smo IConfiguration i spremili ga.
- Da bismo pročitali vrijednost, koristimo `_config["Ključ"]`. Ključevi za nested JSON koriste `:` kao delimiter. Tako `"AppSettings:ApiName"` dohvata vrijednost `"ProductsAPI"` (sekcija **AppSettings** pa onda **ApiName**).
- Sastavili smo string od naziva i verzije i vraćamo ga.

Ako se otvori GET /api/info/version, trebalo bi vratiti:

```
"ProizvodiAPI - Verzija 1.0"
```

Naravno, ako te vrijednosti postoje u appsettings.json.

Tako se može dohvatiti bilo koja pojedinačnu vrijednost. Za connection string postoji posebna metoda `GetConnectionString(name)` koja zapravo traži u sekciji "ConnectionStrings", to smo koristili ranije.

2. Options pattern

Ovo je nešto napredniji pristup gdje definiramo klasu koja odgovara strukturi postavki, npr.:

```
public class AppSettingsOptions {
    public string ApiName { get; set; }
    public string Version { get; set; }
}
```

i tada u Program.cs:

```
builder.Services.Configure<AppSettingsOptions>
(_config.GetSection("AppSettings"));
```

Pa se injektira `IOptions<AppSettingsOptions>` u konstruktor kontrolera i dobije se konfiguirana instanca. Time se dobije jaka tipizacija i validacija, ukoliko je potrebno. No, za par jednostavnih vrijednosti često je overkill, **stoga ćemo ostati na jednostavnijem pristupu.**

Korištenje konfiguracije u praksi

Važno pravilo je držati konfiguracijske vrijednosti **izvan koda** kad god je to nešto što se razlikuje po okolini. Connection string je klasičan primjer (drugačiji za dev, test, prod). Također API ključevi, tajne, ili općenito "magični brojevi". Umjesto da u kodu piše `UseSqlite("Data Source=products.db")`, bolje je povući string iz configa. Umjesto da ime aplikacije bude hardkodirano, bolje je tako nešto postaviti u konfiguraciju.

Za Angular developera, analogija su **environment.ts** datoteke – tamo se drže bazni URL-ovi i slično. Appsettings.json je .NET ekvivalent (s tim da ima i mehanizam za različite env profile).

Primjena na naš projekt

Provjerimo imamo li što još prenijeti u config:

- Trenutno, connection string smo već povukli iz configa.
- Možemo staviti i primjerice JWT postavke (tajni ključ, issuer, audience) u config kad se budemo bavili time, umjesto u kodu.
- Sve što mislimo da bi se mijenjalo bez novog kompiliranja aplikacije, spada u konfiguraciju.

Još o appsettings.Development.json

appsettings.Development.json često služi za preklapanje određenih sekcija iz glavnog **appsettings.json**. Na primjer, može sadržavati detaljnije postavke logiranja ili druge prilagodbe specifične za razvojno okruženje.

ASP.NET Core automatski učitava **appsettings.{Environment}.json** ako takva datoteka postoji i ako je definirano odgovarajuće okruženje, primjerice **Development**. To znači da se određene postavke mogu prilagoditi za različita okruženja – npr. u development okruženju možemo postaviti "UseInMemoryDatabase": true za **EF Core**, dok bi u produkciji koristili pravu bazu podataka.

Za sada je dovoljno znati da ovaj mehanizam postoji, ali u našem slučaju, **defaultna konfiguracija će biti dovoljna**.

Zaključak poglavlja

Konfiguracija možda nije najzanimljiviji dio razvoja, ali je ključna za praktičan rad. Sada znate kako pročitati **connection string** iz **appsettings.json** i kako dohvatiti vlastite postavke.

Ako ste dodali **InfoController**, provjerite vraća li ispravne vrijednosti kako bi bili sigurni da sve radi kako treba.

Ako želite bolje razumjeti kako ASP.NET Core upravlja **konfiguracijom**, istražite sljedeće:

- **Kako se kombiniraju appsettings.json i appsettings.{env}.json**
- **Kako dohvatiti trenutno okruženje (Development, Production, Staging)**
 - Okruženje se može pročitati pomoću **IWebHostEnvironment** ili varijable okoline **ASPNETCORE_ENVIRONMENT**.
- **IOptions pattern za tipiziranu konfiguraciju**
 - Umjesto da ručno dohvaćate vrijednosti iz konfiguracije, možete koristiti **IOptions<T>**, što omogućuje **automatsko mapiranje postavki u klase**.
 - Ovo je korisno za organiziran i čitljiv kod, posebno kad postoji **više povezanih postavki**.
- **Tajne i sigurna pohrana osjetljivih podataka**
 - Connection stringovi i API ključevi ne bi se trebali nalaziti u **appsettings.json**, pogotovo ako se kod dijeli ili koristi **public repo**.
 - ASP.NET Core omogućuje **User Secrets** i **Environment Variables** za sigurno spremanje osjetljivih podataka.

Eksperimentirajte s **različitim okružnjima**, **IOptions patternom** i **učitavanjem konfiguracije** kako bi bolje razumjeli kako ASP.NET Core dinamički prilagođava aplikaciju.

U sljedećem poglavlju prelazimo na nešto dinamičnije – **autentifikaciju i sigurnost**. Ovo je možda **najzahtjevnije poglavlje u knjizi**, ali i jedno od najvažnijih, jer rijetko koji API ostaje potpuno otvoren za sve korisnike. Fokusirat ćemo se na **JWT autentifikaciju**, koja se često koristi u SPA aplikacijama s **Angularom**.

Tjedan 8: Osnove autentifikacije i sigurnosti Web API-ja

Cilj ovog poglavlja je upoznati se s osnovama sigurnosti Web API-ja, s naglaskom na autentifikaciju (authentication) i autorizaciju (authorization). Naučit ćemo razliku između ta dva pojma i implementirati jednostavan mehanizam autentifikacije pomoću JWT (JSON Web Token), koji je danas standard za komunikaciju između SPA aplikacija (Angular frontend) i REST API-ja (backend).

Autentifikacija vs. autorizacija

Prvo, razjasnimo ključne pojmove:

- **Autentifikacija** je proces provjere identiteta korisnika. Primjer je **prijava (login)** – korisnik unosi svoje vjerodajnice (korisničko ime i lozinku), a sustav potvrđuje da je to zaista taj korisnik.
- **Autorizacija** određuje što autentificirani korisnik smije raditi ili vidjeti. Primjerice, **endpointi označeni [Authorize]** odbacit će zahtjeve korisnika koji nisu prijavljeni ili nemaju odgovarajuća prava.

U kontekstu Web API-ja, najčešće se koristi **Bearer Token autentifikacija**. Nakon uspješne prijave, klijent dobiva **token** (niz znakova) koji predstavlja njegov identitet. Taj token se zatim šalje u **Authorization headeru** pri svakom narednom zahtjevu:

```
Authorization: Bearer <token>
```

API provjerava token – je li valjan, nije li istekao, je li ispravno potpisano – i na temelju toga dopušta ili odbija pristup. Ako token nije ispravan, vraća **401 Unauthorized**.

JWT (JSON Web Token)

JWT je specifičan oblik tokena koji je **samodostatan (self-contained)** – unutar njega su zapisane informacije o korisniku i eventualno njegovim pravima. JWT je **Base64 kodiran string** koji se sastoji od tri dijela:

1. **Header** – Metapodaci (npr. algoritam za potpisivanje).
2. **Payload** – Podaci o korisniku (npr. korisničko ime, uloge).
3. **Signature** – Kriptografski potpis koji osigurava da JWT nije mijenjan.

 Ne moramo ručno obrađivati JWT – koristit ćemo **gotove biblioteke** koje će se pobrinuti za generiranje i validaciju tokena.

Također, **nije nužno u potpunosti razumjeti sav kod koji slijedi**. Važnije je shvatiti koncept sigurnosti i što se događa s HTTP zahtjevima.

Cilj je razumjeti osnovne principe autentifikacije i autorizacije, a ne ulaziti u svaki detalj implementacije. Kroz praksu, stvari će postati jasnije!

Postavljanje JWT autentifikacije u ASP.NET Core

Potrebni koraci za implementaciju JWT su:

1. Dodavanje paketa za JWT tokene
2. Konfiguracija aplikacije da podržava tokene
3. Endpoint za dohvatanje tokena kojeg ćemo dalje koristiti u aplikaciji

Dodavanje paketa: Potreban nam je Microsoft.AspNetCore.Authentication.JwtBearer paket

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Konfiguracija JWT u Program.cs: Slijedi dodavanje autentifikacijskog servisa i definiranje parametara tokena:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

// ... unutar builder.Services konfiguracije:
var jwtKey = "OvoJeTajniKljucZaJWT123!!abcabcabc123"; // treba biti dovoljno dugačak random

builder.Services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme =
    JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
    JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options => {
    options.RequireHttpsMetadata = false;
    options.SaveToken = true;
    options.TokenValidationParameters = new TokenValidationParameters {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "moja-aplikacija",
        ValidAudience = "moja-aplikacija-korisnici",
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
    };
});
```

Objašnjenje konfiguracije JWT autentifikacije

Prilikom konfiguracije autentifikacije u ASP.NET Core API-ju koristimo JWT Bearer autentifikaciju, a evo što pojedine postavke znače:

1. Postavljamo defaultnu autentifikacijsku shemu na **JwtBearer**

- Time ASP.NET Core zna da će se autentifikacija provoditi putem JWT tokena.

2. **RequireHttpsMetadata = false**

- Ova postavka omogućuje testiranje API-ja preko običnog HTTP protokola.
- U produkciji bi trebala biti true, jer tokeni trebaju prolaziti samo kroz HTTPS kako bi se osigurala sigurnost.

3. **SaveToken = true**

- Ako API kasnije treba dohvatiti sam token (npr. za logging ili dodatne provjere), omogućava mu da ga pohrani u memoriji.

4. **TokenValidationParameters** – pravila za provjeru tokena:

- **ValidateIssuer** = true – provjerava tko je izdao token.
- **ValidateAudience** = true – provjerava tko je krajnja publika tokena.
 - Mi postavljamo da Issuer = "moja-aplikacija", a Audience = "moja-aplikacija-korisnici".
 - Ovo sprječava korištenje tokena u aplikacijama kojima nisu namijenjeni.
- **ValidateLifetime** = true – provjerava je li token istekao.
 - Token mora sadržavati "exp" claim (vrijeme isteka).
- **ValidateIssuerSigningKey** = true – provjerava je li token digitalno potpisana.
 - Tokeni koji nisu ispravno potpisani bit će odbijeni.

5. **IssuerSigningKey** – ključ za potpisivanje tokena:

- Koristimo simetrični ključ kako bismo osigurali da API može generirati i provjeriti JWT token.
- Ovaj ključ se obično spremi u konfiguraciju (npr. appsettings.json) umjesto da bude hardkodiran u kodu.
- Važno: Ključ mora biti dovoljno dug i siguran kako bi se spriječilo njegovo pogađanje ili dešifriranje.

Integracija autentifikacije u middleware pipeline

Nakon što pozovemo `builder.Build();`, potrebno je osigurati da se autentifikacija i autorizacija pravilno provode prilikom obrade zahtjeva. To se postiže dodavanjem sljedećih middlewarea u pipeline:

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

Pravilni redoslijed

- **app.UseAuthentication();** – prvo provjerava je li zahtjev poslan s valjanim JWT tokenom.
- **app.UseAuthorization();** – zatim provjerava ima li autenticirani korisnik odgovarajuća prava pristupa određenim resursima.

Redoslijed je važan!

Ako `UseAuthorization()` dođe prije `UseAuthentication()`, API neće znati tko je korisnik i pravila autorizacije neće imati učinka.

Generiranje JWT tokena

Da bi autentifikacija imala smisla, trebamo **endpoint za prijavu** (`/api/auth/login`) koji će:

1. Primiti korisničko ime i lozinku.
2. Provjeriti vjerodajnice (za jednostavnost, možemo koristiti hardkodiranog korisnika ili jednostavnu tablicu).
3. Ako su ispravni, generirati JWT token i vratiti ga korisniku.

Stvorimo novi AuthController:

```

using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.IdentityModel.Tokens;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;

[ApiController]
[Route("api/[controller]")]
public class AuthController : ControllerBase {
    private readonly string _jwtKey = "OvoJeTajniKljucZaJWT123!abcabca123";

    // samo radi vježbe hardkodiramo korisnika radi jednostavnosti.
    private readonly string _validUsername = "admin";
    private readonly string _validPassword = "lozinka";

    [AllowAnonymous]
    [HttpPost("login")]
    public IActionResult Login([FromBody] LoginRequest request) {
        if (request.Username == _validUsername && request.Password == _validPassword) {

            // vjerodajnice su ispravne, generiraj token
            var tokenHandler = new JwtSecurityTokenHandler();
            var keyBytes = Encoding.UTF8.GetBytes(_jwtKey);
            var tokenDescriptor = new SecurityTokenDescriptor {
                Subject = new ClaimsIdentity(new[] {
                    new Claim(ClaimTypes.Name, request.Username)
                }),
                Expires = DateTime.UtcNow.AddHours(1),
                Issuer = "moja-aplikacija",
                Audience = "moja-aplikacija-korisnici",
                SigningCredentials = new SigningCredentials(
                    new SymmetricSecurityKey(keyBytes),
                    SecurityAlgorithms.HmacSha256Signature)
            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            string tokenString = tokenHandler.WriteToken(token);
            return Ok(new { token = tokenString });
        }

        // Neuspješna prijava
        return Unauthorized();
    }
}

public class LoginRequest {
    public string Username { get; set; }
    public string Password { get; set; }
}

```

Ovdje se događa dosta toga, pa razjasnimo ključne dijelove:

1. Endpoint /api/auth/login i [AllowAnonymous]

- **[AllowAnonymous]** omogućuje da svi korisnici mogu pristupiti ovom endpointu, čak i ako ostatak API-ja bude zaštićen pomoću **[Authorize]**.
- To je nužno jer se korisnici moraju moći prijaviti prije nego što dobiju pristup ostalim resursima.

2. Provjera korisnika

- U stvarnoj aplikaciji, ovdje bi se korisnički podaci dohvaćali iz baze, a lozinka bi bila hashirana.
- U ovom jednostavnom primjeru, provjeravamo korisnika pomoću obične string usporedbe:

```
if (model.Username == "admin" && model.Password == "password")
```

Ovo je samo za demonstraciju – u produkciji se lozinke nikad ne spremaju u običnom tekstu!

3. Generiranje JWT tokena

Ako su vjerodajnice ispravne, API generira JWT token. Ključni koraci u tom procesu:

- **JwtSecurityTokenHandler**
 - Klasa iz System.IdentityModel.Tokens.Jwt koja omogućuje rad s JWT tokenima.
- **keyBytes**
 - Tajni ključ (IssuerSigningKey), koji se koristi za potpisivanje i provjeru tokena.
 - Mora biti sigurno pohranjen (npr. u **konfiguraciji**, a ne hardkodiran).
- **ClaimsIdentity**
 - Sadrži podatke (claims) koji su zapisani unutar tokena.
 - Obavezno dodajemo barem **korisničko ime**:

```
new Claim(ClaimTypes.Name, username)
```

Ako želimo autorizaciju prema ulogama, možemo dodati **role claim**:

```
new Claim(ClaimTypes.Role, "Admin")
```

- **Expires**
 - Postavlja **vrijeme isteka tokena** (u ovom slučaju 1 sat).
- **Issuer i Audience**
 - Moraju se poklapati s vrijednostima u **TokenValidationParameters** kako bi ASP.NET Core prepoznao token kao valjan.
- **SigningCredentials**
 - Koristimo **simetrični ključ** i algoritam **HMAC SHA256** za potpisivanje tokena.

Kreiranje i slanje tokena klijentu

- `tokenHandler.CreateToken(tokenDescriptor)` stvara token objekt.
- `WriteToken(token)` pretvara ga u string i šalje klijentu u JSON formatu:

```
{ "token": "eyJhbGciOiJ...etc" }
```

4. Ako su vjerodajnice neispravne

- API vraća **HTTP 401 Unauthorized**.

5. Kako klijent koristi token?

Kada klijent (npr. Angular aplikacija) primi token:

1. **Pohranjuje ga** (obično u localStorage ili memoriji).
2. **Šalje ga u svakom zahtjevu prema API-ju**, dodajući ga u **Authorization header**:

```
Authorization: Bearer <token>
```

Sada imamo funkcionalan login endpoint i osnovni JWT sustav autentifikacije. U nastavku ćemo vidjeti kako zaštititi API i koristiti ove tokene za kontrolu pristupa.

Zaštita endpointa pomoću [Authorize]

Na strani Web API-ja, možemo zaštititi određene **akcije** ili **čitave kontrolere** tako da zahtijevaju valjan JWT token. To radimo dodavanjem **[Authorize]** atributa.

Primjer scenarija:

- **Čitanje proizvoda (GET)** treba biti **dostupno svima**.
- **Dodavanje ili brisanje proizvoda (POST, DELETE)** smije obavljati **samo autentificirani korisnik**.

To postižemo kombiniranjem **[Authorize]** i **[AllowAnonymous]**:

```
[HttpGet]
[AllowAnonymous] // omogućujemo svima da čitaju proizvode
public async Task<ActionResult<IEnumerable<Product>>> GetAll() { ... }

[HttpPost]
[Authorize] // samo prijavljeni korisnik (s valjanim tokenom) može dodati
public async Task<ActionResult<Product>> Create(Product newProduct) { ... }
```

Kako [Authorize] funkcioniра?

- **[Authorize] iznad metode** – zahtjev mora sadržavati JWT token, inače API vraća **401 Unauthorized**.
- **[AllowAnonymous] iznad metode** – dopušta svima pristup bez autentifikacije.
- **[Authorize] iznad kontrolera** – **sve metode postaju zaštićene**, osim ako pojedinačno označimo neke s **[AllowAnonymous]**.

Ako pokušamo pozvati **zaštićeni endpoint bez tokena** ili s **neispravnim tokenom**, ASP.NET Core automatski vraća **401 Unauthorized** prije nego što kontroler uopće obradi zahtjev.



Napomena: Obavezno testiraj:

Bez tokena: dobij 401.

S tokenom: uspješan odgovor. To možete testirati preko Postmana ili Swagger (Swagger UI ima mogućnost unosa Bearer tokena za testiranje ali ga se mora podesiti – u slijedećem poglavljju pokazati će se konfiguracija za to).

Sigurnosni savjeti pri radu s JWT autentifikacijom

Kako biste osigurali da vaša aplikacija koristi **sigurnu i pravilno postavljenu JWT autentifikaciju**, imajte na umu sljedeće preporuke:

1. Nemojte hardkodirati tajne u kodu

- U **demo primjerima** koristili smo hardkodirane ključeve, ali to **nikad ne radite u produkciji**.
- Umjesto toga, koristite **appsettings.json** i dohvaćajte ih kroz konfiguraciju:

```
"Jwt": {  
  "Key": "tajni_kljuc",  
  "Issuer": "moja-aplikacija",  
  "Audience": "moja-aplikacija-korisnici"  
}
```

- Još bolja praksa u **produkciji** je koristiti **environment varijable** ili **tajne servise** poput **Azure Key Vault**, AWS Secrets Manager, HashiCorp Vault itd.

2. Koristite HTTPS u produkciji

- **Nikad nemojte slati JWT preko plain HTTP-a**, jer može biti presretnut u **MITM (Man-In-The-Middle)** napadu.
- U developmentu možete koristiti `RequireHttpsMetadata = false` za testiranje, ali **u produkciji HTTPS mora biti obavezan**.

3. Isteč tokena

- Trenutno smo postavili **token da vrijedi 1 sat**, što je razumno za većinu aplikacija.
- U produkciji biste možda htjeli koristiti **refresh tokene** kako bi korisnici mogli ostati prijavljeni dulje vrijeme bez potrebe za ponovnim loginom.
- **Važno:** Nemojte postavljati **beskonačno trajanje tokena**, jer to povećava sigurnosni rizik.

4. JWT payload je vidljiv

- JWT **nije šifriran**, već samo **potpisani**.
- To znači da **bilo tko može dekodirati token** pomoću **base64 dekodera** i vidjeti sadržaj.
- Primjer JWT dekodera: jwt.io
- **Nikad nemojte stavljati osjetljive podatke** u JWT claimove (npr. lozinke, osobne podatke).

Autorizacija po ulogama (Role Based Authorization)

JWT token može sadržavati **claim za ulogu** (npr. "role": "Admin").

ASP.NET Core omogućuje **ograničavanje pristupa endpointima** pomoću `[Authorize(Roles="Admin")]`.

Kako dodati role u JWT token?

Pri generiranju tokena, dodajte **role claim**:

```
new Claim(ClaimTypes.Role, "Admin")
```

Kako omogućiti autorizaciju po ulogama?

U `AddJwtBearer` konfiguraciji (`TokenValidationParameters`), dodajte:

```
RoleClaimType = ClaimTypes.Role
```

Zatim, u API-ju zaštitite endpoint primjerice **samo za administratore**:

```
[Authorize(Roles = "Admin")]
[HttpGet]
public async Task<ActionResult<List<User>>> GetUsers() { ... }
```

Time osiguravamo da samo korisnici s ulogom "Admin" mogu pristupiti ovom endpointu.

Zaključak poglavlja

U ovom poglavlju obradili smo dosta novih informacija i napisali značajnu količinu koda.

Ukratko, sada imamo **JWT autentifikaciju** postavljenu na našem Web API-ju i zaštitili smo određene endpoint-e. Time smo stvorili **full-stack mini aplikaciju** koja uključuje:

- ✓ **C# backend** s bazom podataka
- ✓ **Sigurnost putem JWT autentifikacije**
- ✓ **Pripremljeni API koji frontend (npr. Angular) može koristiti**

U praktičnoj primjeni, **Angular frontend** bi u svakom HttpClient zahtjevu dodavao **Authorization header** s tokenom kada poziva zaštićene endpoint-e, primjerice:

```
Authorization: Bearer <token>
```

Tako API zna tko je korisnik i može mu dopustiti ili odbiti pristup.

Ako je **generiranje JWT tokena** djelovalo zbumujuće, ne brinite – u praksi se taj dio često **preuzme iz primjera i prilagodi** potrebama aplikacije. Važno je razumjeti osnovni koncept:

- **Token je digitalno potpisani** i provjerava se pomoću server-side ključa.
- **Sadrži identitet korisnika** (npr. korisničko ime, uloge) i API koristi te podatke za autorizaciju.

Što slijedi?

U sljedećem poglavlju napravit ćemo **mini-projekt** u kojem ćemo integrirati sve što smo naučili.

- Proći ćemo ponovno kroz **cijeli proces izrade API-ja** – od početka do kraja.
- Iskoristit ćemo dio koda koji već imamo, ali cilj je da kroz **praktičnu vježbu** sve sjedne na svoje mjesto.

S ovim znanjem, naš API je sada **spreman za upotrebu u stvarnim aplikacijama!**

Tjedan 9: Izrada mini projekta: kompletan Web API

Cilj ovog poglavlja je primijeniti sve što smo naučili kroz izgradnju malog projekta od nule – od pokretanja novog Web API projekta, preko modeliranja baze i kontrolera, do implementacije sigurnosti i korisnih dodataka. Ovaj projekt će poslužiti kao rekapitulacija i prilika za utvrđivanje stečenog znanja.

Kao završnu vježbu, izradit ćemo **Task Manager API** – jednostavan **REST API** za upravljanje zadacima (to-do listom). Ovaj mini-projekt će objediniti sve što smo naučili do sada i omogućiti nam da prođemo kroz cijeli proces izrade API-ja od početka do kraja.

Što ćemo napraviti?

- Definirat ćemo **entitet "Task" (Zadatak)** s nekoliko svojstava.
- Omogućit ćemo **CRUD operacije** (kreiranje, dohvaćanje, ažuriranje i brisanje zadataka).
- Opcionalno ćemo zaštитiti određene operacije **JWT autentifikacijom** – npr. samo prijavljeni korisnici moći će dodavati i brisati zadatke.
- Ako već nismo, **integrirat ćemo Swagger dokumentaciju** kako bismo dobili interaktivno sučelje za testiranje API-ja. (Swagger je često već uključen po defaultu u ASP.NET Core Web API projektima).

Ovim projektom ćemo završiti naše putovanje kroz izradu API-ja, osigurati da sve naučeno možemo praktično primjeniti i stvoriti solidnu osnovu za daljnji razvoj **full-stack aplikacija**.

Postavljanje projekta i modela

Kreirajmo projekt:

```
dotnet new webapi --use-controllers -o TaskApi
```

(ili ručno u Visual Studiju).

Otvorimo projekt i provjerimo da li TaskApi.csproj ima reference na **Swashbuckle** (Swagger). Ako ne, dodajemo ih:

```
dotnet add package Swashbuckle.AspNetCore
```

Definirati čemo model zadatka: Dodajemo novu klasu `Models/TaskItem.cs`:

```
public class TaskItem {  
    public int Id { get; set; }  
    public string Title { get; set; }  
    public string Description { get; set; }  
    public bool IsCompleted { get; set; }  
    public DateTime? DueDate { get; set; }  
}
```

Ova klasa predstavlja jedan zadatak: Id, naslov, opis, flag je li dovršen, i opcionalni rok (DueDate).

Kreiranje DbContext za bazu

Dodajemo novu AppDbContext klasu (može u root projekta ili u Data folder):

```
using Microsoft.EntityFrameworkCore;
public class AppDbContext : DbContext {
    public AppDbContext(DbContextOptions<AppDbContext> options) :
    base(options) { }
    public DbSet<TaskItem> Tasks { get; set; }
}
```

i Prije toga naravno trebamo dodati referencu na EF Core, objašnjeno u prethodnim poglavljima.

Registrirajmo DbContext u Program.cs: Otvoriti ćemo Program.cs, i dodati

```
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlite(
builder.Configuration.GetConnectionString("DefaultConnection") ?? "Data
Source=tasks.db"));
```

Tako koristimo connection string "DefaultConnection" iz appsettings.json, ili fallback na "tasks.db" ako nije tamo definiran. Naravno, moramo dodati i `using Microsoft.EntityFrameworkCore;` na vrh.

Connection string u appsettings.json:

U appsettings.json dodajemo

```
"ConnectionStrings": {
    "DefaultConnection": "Data Source=tasks.db"
}
```

EF Core migracija: Pokrenuti ćemo migraciju s kojom ćemo stvoriti bazu i tablicu

```
dotnet ef migrations add InitTasks
```

```
dotnet ef database update
```

Task Controller (CRUD logika)

Dodajemo novi kontroler `Controllers/TasksController.cs`:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace TaskApi.Controllers {
    [ApiController]
    [Route("api/[controller]")]
    public class TasksController : ControllerBase {
        private readonly AppDbContext _context;
        public TasksController(AppDbContext context) {
            _context = context;
        }

        // GET: api/tasks
        [HttpGet]
        public async Task<ActionResult<IEnumerable<TaskItem>>> GetAll() {
            var tasks = await _context.Tasks.ToListAsync();
            return Ok(tasks);
        }

        // GET: api/tasks/{id}
        [HttpGet("{id}")]
        public async Task<ActionResult<TaskItem>> GetById(int id) {
            var task = await _context.Tasks.FindAsync(id);
            if (task == null) return NotFound();
            return Ok(task);
        }

        // POST: api/tasks
        [Authorize] // zahtijeva autentikaciju za kreiranje zadatka
        [HttpPost]
        public async Task<ActionResult<TaskItem>> Create(TaskItem task) {
            _context.Tasks.Add(task);
            await _context.SaveChangesAsync();
            return CreatedAtAction(nameof(GetById), new { id = task.Id }, task);
        }
    }
}
```

```
// PUT: api/tasks/{id}
[Authorize]
[HttpPut("{id}")]
public async Task<IActionResult> Update(int id, TaskItem taskData) {
    if (id != taskData.Id) return BadRequest();
    var task = await _context.Tasks.FindAsync(id);
    if (task == null) return NotFound();
    // Ažuriramo polja
    task.Title = taskData.Title;
    task.Description = taskData.Description;
    task.IsCompleted = taskData.IsCompleted;
    task.DueDate = taskData.DueDate;
    await _context.SaveChangesAsync();
    return NoContent();
}

// DELETE: api/tasks/{id}
[Authorize]
[HttpDelete("{id}")]
public async Task<IActionResult> Delete(int id) {
    var task = await _context.Tasks.FindAsync(id);
    if (task == null) return NotFound();
    _context.Tasks.Remove(task);
    await _context.SaveChangesAsync();
    return NoContent();
}
}
```

💡 Napomene:

Ovdje smo odlučili da kreiranje, ažuriranje i brisanje zadataka zahtijevaju da korisnik bude autentificiran (Authorized). Čitanje (GET) ostaje otvoreno.

Kod je jako sličan onom za Products ranije, samo prilagođen TaskItem entitetu.

Autentikacija (ponovno) i korisnici

Kako smo stavili `[Authorize]` na par akcija, moramo omogućiti login. Možemo iskoristiti isti AuthController koji smo ranije napisali i možda ga malo prilagoditi za "TaskApi" naziv:

- Kopirati ćemo AuthController kod iz poglavlja 8 (Login action, hardkodirani korisnik).
- Postaviti ćemo `[jwtKey]` iz configa: bolje nego hardkodiran, stoga dodajemo u appsettings vrijednosti:

```
"JwtSettings": {
  "Key": "NekiDugiNasumicniTajniKljuc1234567890",
  "Issuer": "TaskAPI",
  "Audience": "TaskAPIUsers"
}
```

U Program.cs promjeniti ćemo:

```
var jwtSection = builder.Configuration.GetSection("JwtSettings");
var jwtKey = jwtSection.GetValue<string>("Key");
var issuer = jwtSection.GetValue<string>("Issuer");
var audience = jwtSection.GetValue<string>("Audience");

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = issuer,
            ValidAudience = audience,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
    });
});
```



U AuthController, nećemo hardkodirati key nego koristiti isti key/issuer:

Injektirati ćemo `IConfiguration`, pa izvući ključ.

U pipeline (Program.cs) dodati ćemo `app.UseAuthentication();` i `app.UseAuthorization();`.

Swagger UI i testiranje

Swagger (Swashbuckle) generira UI na `/swagger`. Tamo ćemo vidjeti sve naše endpointe. Možemo testirati:

- Bez JWT tokena, GET `/api/tasks` radi, POST `/api/tasks` vraća 401.
- Za test JWT-a u Swaggeru: pri vrhu UI-a ima gumb "Authorize". Kliknuti ćemo na njega i unijeti "`Bearer {token}`" (s Bearer prefixom).

Da bi se ovaj gumb vidi, potrebno je i konfigurirati swagger da podržava gumb za unos tokena. To se definira unutar `Program.cs`

```
builder.Services.AddSwaggerGen(c => {
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme {
        In = ParameterLocation.Header,
        Description = "JWT Authorization header using the Bearer scheme.",
        Name = "Authorization",
        Type = SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });
    c.AddSecurityRequirement(new OpenApiSecurityRequirement {
        {
            new OpenApiSecurityScheme {
                Reference = new OpenApiReference {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            new List<string>()
        }
    });
});
```

Taj kod kaže swaggeru da ima Bearer Auth i prikaže Authorize modal.

- Uzeti ćemo token preko `/api/auth/login` (možemo u swaggeru taj poziv napraviti unoseći login podatke u body), kopirati token, zalijepiti u Authorize. Sada Swagger UI ima token u headeru za naredne pozive.
- Sada možemo probati POST/PUT/DELETE Task - trebalo bi proći ako token valja.

Time smo prošli kompletan razvojni ciklus:

- Postavili projekt
- Modelirali entitet i bazu
- Napravili CRUD kontroler
- Dodali autentifikaciju i autorizaciju
- Testirali putem Swagger UI (ili možeš i Postmanom ako želiš).

Dodatno: NuGet paketi i biblioteke

U projektu smo koristili:

- Entity Framework Core (nuget)
- Swashbuckle (Swagger UI)
- JWT Bearer (Microsoft.AspNetCore.Authentication.JwtBearer)
- To su primjeri korištenja paketa (koji smo dodali putem dotnet CLI). U .NET ekosustavu, NuGet paketi su uobičajeni način širenja funkcionalnosti, analogno npm paketima u Node/Angular svijetu.

Primjer još jednog korisnog paketa mogao bi biti **Serilog** za logiranje akcija ili **AutoMapper** za mapiranje objekata (npr. entiteta u DTO i obratno). No, za naš scope, to ostaje kao "istraži dalje".

Zaključak poglavlja

Ako ste slijedili sve korake, sada imate funkcionalan **Task Manager API** koji omogućuje registriranim korisnicima da kreiraju i upravljaju svojim zadacima!



Iako smo implementirali osnovne funkcionalnosti, **nismo povezali zadatke s korisnicima**, što bi zahtjevalo dodatni korisnički model. To bi moglo uključivati:

- **Spremanje userId u Task entitet**
- **Integraciju s ASP.NET Identity** ili ručno upravljanje korisnicima

To može biti **odličan sljedeći korak** za proširenje projekta i vježbanje naprednijih koncepata.

Što je najvažnije, kroz ovaj praktični primjer prošli ste **cijeli proces razvoja API-ja**, od definiranja modela i baza podataka do autentifikacije i autorizacije. Tako ste učvrstili svoje znanje i vjerojatno uočili gdje imate još prostora za usavršavanje.

Ako nešto nije radilo iz prve, **debugiranje i rješavanje problema** samo su dodatno iskustvo koje će vam koristiti u budućem radu.

U finalnom poglavlju napraviti ćemo kratak **pregled svega što smo prošli** i dati **smjernice za daljnje učenje i razvoj vještina**.

Tjedan 10: Pregled, dorada i sljedeći koraci

Cilj ovog poglavlja je sažeti najvažnije što smo naučili, istaknuti moguće propuste koje treba ispraviti i dati smjernice za daljnje učenje kako bi se znanje o .NET Web API-ju produbilo. Ukratko, ovo poglavlje služi kao retrospektiva i plan za daljnji razvoj vještina.

Kratki pregled naučenog

Prošli smo dosta toga u relativno kratkom vremenu:

- ✓ Postavljanje .NET okruženja i osnove C# sintakse – naučili smo da je sličan TS/JS-u u mnogim aspektima i brzo smo krenuli na konkretnе primjere.
- ✓ Objektno orijentirano programiranje u C# – klase, objekti, nasljeđivanje, interfejsi. Shvatili smo da Angular developerske vještine ovdje dobro dođu, jer koncepti su paralelni.
- ✓ ASP.NET Core Web API osnove – struktura projekta, Program.cs, kontroleri i mehanizam ruta, kako se HTTP zahtjev obrađuje.
- ✓ Kreirali smo prvi kontroler s CRUD operacijama koristeći in-memory listu – praktična vježba mapiranja HTTP metoda na logiku servera i vraćanja odgovarajućih odgovora (200 OK, 404 NotFound, itd.).
- ✓ Dependency Injection – jedna od ključnih značajki modernog .NET-a. Vidjeli smo kako injektirati servise (npr. vlastiti TimeService, DbContext, konfiguraciju, itd.) i kako to čini kod modularnijim i testabilnijim.
- ✓ Integracija baze podataka preko Entity Framework Core – veliki korak gdje smo uveli trajnu pohranu. Naučili smo kreirati DbContext, modele, povezati se s bazom (SQLite u našem slučaju) i koristiti migracije da generiramo bazu iz koda.
- ✓ Konfiguracija i appsettings – kako držati connection string i druge postavke izvan koda i dohvatiti ih kroz IConfiguration. To je važan aspekt za realne projekte (razdvajanje konfiguracije i koda).
- ✓ Autentifikacija i sigurnost – uveli smo JWT autentifikaciju kao primjer modernog pristupa za API-je. Sada razumiješ osnovnu razliku između autentikacije i autorizacije i imaš predodžbu kako tokeni funkcioniraju.
- ✓ Mini projekt – praktično objedinjavanje svega: od nule do funkcionalnog API-ja. To je simuliralo realan scenarij razvoja manje aplikacije.

To je stvarno puno za Angular developera koji je tek zakoračio u .NET! Ako ste uspješno pratili, svaka čast – proširili ste svoje full-stack sposobnosti. 

Dorada i najbolje prakse

Prije nego što završimo s našim **Task Manager API-jem**, vrijedi razmisliti o mogućnostima dorade i poboljšanja kako bi API bio robusniji, sigurniji i lakši za održavanje.

1. Validacija ulaznih podataka

Trenutno **nemamo nikakvu validaciju** podataka koje korisnik šalje. .NET omogućuje dodavanje **ugrađenih atributa za validaciju** (poput [Required] i [MaxLength]) kako bi API automatski spriječio loše podatke i vratio jasne poruke o greškama. Ovo bi bilo korisno za polja poput **naslova i opisa zadatka**, osiguravajući da su ispravni prije nego što ih API obradi.

2. Bolje rukovanje greškama i izuzecima

Trenutno koristimo ručne provjere (if-statementi) i vraćamo NotFound ili BadRequest kada je potrebno. Ovo je dobar početak, ali bolje rješenje bilo bi koristiti **globalni middleware za rukovanje izuzecima**, koji bi automatski presreo neočekivane greške i poslao korisniku jasnu poruku umjesto neprikladnih stack traceova. U .NET-u postoje ugrađene opcije poput UseExceptionHandler ili se može napisati vlastiti middleware za upravljanje iznimkama.

3. Razdvajanje slojeva koda

U malim projektima, poslovna logika se često nalazi unutar kontrolera, ali kako aplikacija raste, postaje korisno razdvojiti je u **servise i/ili repozitorije**. Na taj način API ostaje modularan, čišći i lakši za testiranje. U našem slučaju, mogli bismo dodati poseban servis za rad s podacima o zadacima, a kontroler bi bio odgovoran samo za obradu HTTP zahtjeva i odgovora.

4. Dokumentacija API-ja

Swagger već pruža automatski generiranu dokumentaciju, ali dodatni **README fajl** mogao bi pomoći budućim korisnicima ili developerima da razumiju projekt. README bi trebao sadržavati informacije o **instalaciji, pokretanju, konfiguraciji i primjerima API zahtjeva**.

5. Poboljšanje Swagger dokumentacije

Osim osnovne Swagger podrške, bilo bi korisno dodati **detaljne XML komentare** u kod, koji bi bili uključeni u Swagger dokumentaciju. To bi omogućilo da Swagger prikazuje jasne opise API endpointa, parametara i povratnih vrijednosti.

 Ovim projektom prošli smo kroz ključne koncepte izrade Web API-ja u ASP.NET Core-u, uključujući rad s bazom podataka, autentifikaciju i zaštitu endpointa. Iako je API funkcionalan, postoji puno prostora za doradu i prilagodbu ovisno o potrebama.

Samoprovjera znanja

Evo par stvari koje bi sada mogli moći objasniti (čak i nekom drugom, ili samom sebi naglas, vjerujte, pomaže):

- Kako kreirati novi ASP.NET Core Web API projekt i pokrenuti ga.
- Gdje se podešava routing i kako dodati novi endpoint.
- Kako izgleda tipičan kontroler i kako HTTP GET/POST mapira na metode.
- Kako povezati bazu i što je to DbContext i migracija.
- Što je dependency injection i kako injektirati npr. kontekst baze u kontroler.
- Što je JWT token i kako otprilike funkcionira login s tokenom.
- Kako zaštititi rute s [Authorize] i čemu služi [AllowAnonymous].
- Gdje se drže konfiguracije poput connection stringa i kako ih pročitati.

Također, očekivalo bi se da razumijete i sljedeće pojmove:

- **Varijable i tipovi podataka** (int, string, bool, decimal, DateTime)
- **Petlje i uvjetne naredbe** (if, switch, for, while, foreach)
- **Osnovne kolekcije** (List<T>, Dictionary<K,V>, Array)
- **Osnovne operacije nad kolekcijama** (Add, Remove, Count, Where, Select)
- **Metode i parametri** (razlika između void i return, opcionalni parametri, overload)
- **Osnovne klase i objekti** (kako napraviti klasu i instancirati objekt)
- **Što su interfejsi (interface) i kako ih koristiti**
- **Osnovne LINQ operacije** (Where, Select, OrderBy, FirstOrDefault)
- **HTTP metode i statusni kodovi** (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 404 Not Found)
- **Osnove rada s JSON podacima** i kako ih Web API vraća kao odgovor

Ovo su osnovne stvari koje će pomoći da se lakše snađete u **C# i Web API svijetu**. Ako nešto od ovoga nije jasno, **zastanite, istražite i eksperimentirajte s kodom** – najbolje se uči kroz praksu

Sljedeći koraci (što dalje učiti)

Sada kada imate **solidnu osnovu** u radu s ASP.NET Core Web API-jem, postoje mnogi smjerovi u kojima možete **nadograditi svoje znanje**. Ovisno o tome gdje se želite usmjeriti, možete dublje istražiti sljedeće teme:

LINQ i napredniji EF Core

- **LINQ** je moćan alat za filtriranje, projiciranje i agregaciju podataka. Ako koristite **RxJS** u Angularu, LINQ će ti se također svidjeti jer omogućuje **deklarativno oblikovanje podataka**.
- Vrijedi istražiti **lambda izraze** i metode kao što su Where, Select, OrderBy, GroupBy, Any, All.
- **Entity Framework Core** također ima puno naprednih mogućnosti, poput **relacija između entiteta, lazy loadinga, transakcija i optimizacije upita**.

Asinkrono programiranje u .NET-u

- Već smo koristili async/await, ali razumijevanje **što se događa u pozadini** može pomoći da izbjegnete probleme poput **deadlocka** ili neefikasnih asinkronih poziva.
- Naučite **što je Task i ValueTask, kako radi ConfigureAwait(false), kada koristiti Parallel.ForEach umjesto async petlji**.
- Ova znanja su ključna ako budete radili na **visoko performansnim API-jevima**.

Više o autentikaciji i sigurnosti

- **ASP.NET Identity** – ako trebate sustav za **registraciju korisnika, upravljanje lozinkama i ulogama**, ovo je ugrađeno rješenje.
- **Refresh tokeni** – JWT tokeni imaju ograničen vijek trajanja, a refresh tokeni omogućuju automatsko osvježavanje sesije bez potrebe za ponovnom prijavom.
- **OAuth/OpenID Connect** – ako želite omogućiti **prijavu putem Googlea, Microsofta (Azure AD) ili Facebooka**, morate razumjeti ove protokole.

Napredne Web API teme

- **Verzioniranje API-ja** – kada API raste, ponekad je potrebno podržati više verzija (v1, v2).
- **Paging i filtriranje podataka** – korisno kada API vraća velike setove podataka (`?pageSize=10&page=2`).
- **Upload datoteke** – ako API treba primati slike, dokumente, videozapise, vrijedi istražiti kako to najbolje implementirati.
- **Real-time komunikacija** – SignalR omogućuje **real-time push notifikacije** s API-ja na frontend.
- **gRPC i GraphQL** – moderni pristupi koji mogu biti bolji od klasičnog REST-a u određenim slučajevima.

Testiranje u .NET-u

- **Unit testovi** – nauči osnove testiranja servisa pomoću **xUnit ili MSTest**.
- **Integration testovi** – Microsoft nudi **TestServer**, koji omogućuje da pokreneš API u memoriji i testiraš njegove endpoint-e.
- **Mocking** – kako zamijeniti prave servise i baze podataka testnim verzijama (koristeći Moq ili NSubstitute).

Clean Architecture i dizajnerski obrasci

- Kako API raste, organizacija koda postaje bitna. **Clean Architecture** i **Onion Architecture** su popularni načini da se osigura **modularnost i skalabilnost**.
- **Repository pattern** – odvajanje poslovne logike od pristupa podacima.
- **CQRS i MediatR** – razdvajanje **upita i komandi**, korisno za **event-driven arhitekturu**.

Provjerite stečeno znanje u praksi – napredniji WebApi

Ukoliko ste prošli većinu dodatnih koraka, biti ćete spremni za izradu malo naprednijeg WebApi projekta s kojim ćete dodatno utvrditi vaše znanje.

Opis zadatka

Izradite Web API aplikaciju koja upravlja događajima (eventima) i sudionicima. Aplikacija treba omogućiti kreiranje, ažuriranje, brisanje i pregled događaja te registraciju sudionika na događaje. Osim osnovnih CRUD operacija, očekuje se implementacija autentifikacije (npr. JWT) i osnovne autorizacije – tako da samo ovlašteni korisnici mogu, primjerice, kreirati ili uređivati događaje.

Glavni zahtjevi:

1. Entiteti i relacije:

- **Event (Događaj):** Informacije poput naziva, opisa, datuma, lokacije.
- **Participant (Sudionik):** Podaci kao što su ime, e-mail i status (npr. potvrđen, na čekanju).
- **Registracija:** Veza između sudionika i događaja. Svaki događaj može imati više sudionika, a svaki sudionik može biti registriran na više događaja.

2. Autentikacija i autorizacija:

- Implementirajte JWT autentifikaciju.
- Zaštićeni endpointi: samo autentificirani korisnici mogu kreirati, uređivati ili brisati događaje.
- (Opcionalno: možete implementirati i uloge, primjerice "Admin" za upravljanje događajima i "User" za registraciju.)

3. CRUD operacije:

- Endpointi za upravljanje događajima (kreiranje, čitanje, ažuriranje, brisanje).
- Endpointi za upravljanje sudionicima i njihovom registracijom na događaje.
- Endpoint za pretraživanje događaja po datumu, lokaciji ili ključnim riječima.

4. Baza podataka:

- Koristite Entity Framework Core za rad s bazom.
- Napravite migracije za kreiranje potrebnih tablica i relacija.

5. Struktura koda i unit testovi:

- Razmislite o dobroj organizaciji koda – razdijelite slojeve (npr. kontroleri, servisi, repozitoriji) kako bi vaš projekt bio čitljiv i održiv.
- Kreirajte zaseban projekt ili mapu za unit testove. Testirajte ključne dijelove poslovne logike i endpointove, uključujući validaciju podataka, ispravno rukovanje greškama i osnovne CRUD operacije.

6. Naprednija rješenja i dodatne funkcionalnosti (po želji):

- Iako su osnovni zahtjevi navedeni, potičemo vas da istražujete i implementirate dodatne značajke koje nisu obuhvaćene u knjizi. To mogu biti, primjerice, napredna validacija podataka, pagination i sortiranje rezultata, ili čak jednostavno logiranje aktivnosti (tko je kreirao ili ažurirao događaj).

Napomena:

Ovaj zadatak nije samo "uradi i gotovo". Važno je da tijekom izrade zadatka sami istražujete i implementirate naprednija rješenja koja nadilaze sadržaj knjige. Na taj način ćete dodatno produbiti svoje znanje i vještine.

Zaključna riječ

Učiti novu tehnologiju paralelno uz posao i svakodnevne obaveze nije lako, zato si zaslužujete čestitke za napravljen velik korak prema full-stack razvoju. Kombinacija Angular frontenda i .NET backenda čini vas fleksibilnijim developerom koji može kvalitetno rješavati zadatke s obje strane projekta.

.NET svijet je velik, kompleksan i nudi jako puno mogućnosti, ali sada imate čvrstu osnovu na kojoj možeš graditi dalje. Važno je da nastavite redovno istraživati i isprobavati nove stvari – samostalno ili kroz male projekte.

Ako negdje zapnete ili nešto nije jasno, ne brinite – **ChatGPT** i dokumentacija su uvijek pri ruci. Također Microsoft Docs nudi sjajne primjere i *tutoriale*, a vjerujem da već dobro znate koliko i Stack Overflow i razni tehnički blogovi mogu biti korisni.

Sada imate čvrstu podlogu – iskoristite je! Nastavljajte se razvijati, učiti, eksperimentirati, i nemojte se bojati postavljati pitanja.

Želim vam puno uspjeha i zadovoljstva u dalnjem učenju i kodiranju!

Zoran

Dodatak: 100 najčešćih pitanja za junior i mid-level developere

Kako bismo vas potaknuli na daljnje istraživanje i produbljivanje znanja, u ovaj priručnik dodajemo posebno poglavlje sa **100 najčešćih pitanja** iz područja C#-a, ASP.NET Core Web API-ja, LINQ-a, Entity Framework Core-a, Dependency Injectiona i sigurnosti backend aplikacija.

Ova pitanja osmišljena su kako bi vas usmjerila na ključne koncepte i izazove s kojima se možete susresti u svakodnevnom radu. Cilj je potaknuti vas na samostalno istraživanje, čitanje dokumentacije, eksperimentiranje s kodom i praktičnu primjenu naučenog.

Pitanja su prilagođena junior i mid-level backend developerima te mogu poslužiti kao alat za samoprovjeru ili pripremu za tehničke intervjuve. Bez obzira koristite li ih za učenje ili testiranje vlastitog znanja, pomoći će vam u kontinuiranom razvoju vještina i boljem razumijevanju .NET ekosustava.

Za lakšu navigaciju, pitanja su grupirana po temama (osnove, EF Core, LINQ itd.) i sortirana od najjednostavnijih prema najzahtjevnijima.

Osnove C# programskog jezika

- Što su klase, a što objekti u C#? Kako se u kodu kreira instanca (objekt) neke klase?
- Objasni svrhu ključne riječi using u C#. U kojem se slučaju koristi za uvoz prostora imena, a u kojem za automatsko upravljanje resursima?
- Što su svojstva (properties) u C# i kako se razlikuju od polja (fields) unutar klase?
- Što je konstruktor klase u C# i koja je njegova uloga? Po čemu se konstruktor razlikuje od obične metode?
- Koji modifikatori pristupa postoje u C# (npr. public, private, protected, internal) i kako utječu na dostupnost članova klase?
- Objasni razliku između vrijednosnih (value) i referentnih (reference) tipova u C#. Zašto je ta razlika važna?
- Što je delegat (delegate) u C# i za što se koristi? Daj jednostavan primjer njegove upotrebe.
- Objasni sintaksu lambda izraza u C# i u kojim situacijama olakšava pisanje koda. Daj primjer jednostavnog lambda izraza.
- Kako funkcioniraju događaji (events) u C# i koja je njihova veza s delegatima? Navedi primjer gdje bi iskoristio event.
- Kako se definira i koristi statička (static) metoda ili svojstvo u C#? Po čemu se statički članovi razlikuju od instancijskih (nestatičnih)?
- Kako se hvataju i obrađuju iznimke (exceptions) u C#? Opiši korištenje try-catch-finally blokova i kako definirati vlastitu iznimku.
- Što su generički tipovi (generics) u C# i kako doprinose ponovnoj upotrebni koda? Navedi primjer korištenja generičke kolekcije.
- Kako se parametri mogu proslijediti metodi po referenci u C#? Objasni razliku između ključnih riječi ref i out pri prosljeđivanju argumenata.
- Opiši osnovne principe objektno orijentiranog programiranja – nasljedivanje, polimorfizam i enkapsulacija – u kontekstu C# jezika.
- Koja je razlika između apstraktne klase i sučelja (interface) u C#? Kada koristiti apstraktnu klasu, a kada sučelje?
- Što znači da je metoda u C# „preopterećena“ (overloaded) i kako se to razlikuje od „nadjačane“ (overridden) metode? Navedi primjer za svaki slučaj.
- Kako se provodi upravljanje memorijom u .NET-u i koja je uloga garbage collectora u C# aplikacijama?
- Koja je razlika između operatora == i metode Equals() pri usporedbi objekata u C#? U kojim situacijama ta dva načina mogu dati različite rezultate?
- Što su ekstenzijske metode (extension methods) u C# i kako se definiraju? Čemu služe u praksi?
- Koja je razlika između nepromjenjivog tipa string i klase StringBuilder u C#? Kada je bolje koristiti StringBuilder za rad s tekstrom?

Web API u .NET-u (ASP.NET Core Web API)

- Što je ASP.NET Core Web API i čemu služi? Ukratko objasni kako se razlikuje od klasične web-aplikacije (npr. MVC aplikacije).
- Koje korake treba poduzeti da postaviš i pokreneš novi ASP.NET Core Web API projekt na svom računalu?
- Što je kontroler (Controller) u ASP.NET Core Web API-u i kako se definira? Kako otprilike izgleda tipičan kontroler s akcijskim metodama?
- Kako funkcioniра routing u ASP.NET Core Web API-u? Kako se određuju URL putanje (rute) koje mapiraju na određene akcijske metode u kontroleru?
- Na koje se načine može definirati rutiranje u Web API-u: putem atributa [Route] i [HttpGet]/[HttpPost] ili konvencijom? Po čemu se ta dva pristupa razlikuju?
- Navedi glavne HTTP metode (verbe) koje se koriste u Web API-ju. Koja je svrha metoda GET, POST, PUT, DELETE, a kada se koriste i druge poput PATCH ili OPTIONS?
- Kako u kodu kontrolera specificirati koji HTTP verb odgovara pojedinoj akciji? Objasni na primjeru upotrebu atributa poput [HttpGet] i [HttpPost].
- Kako se parametri iz URL-a (route parametri ili query string) te tijela HTTP zahtjeva automatski mapiraju na parametre metode u kontroleru? Opiši ukratko koncept model bindinga.
- Kako Web API vraća odgovore klijentu? Objasni ulogu sučelja IActionResult i kako koristiti metode poput Ok(), BadRequest() ili NotFound() za slanje odgovarajućih HTTP status kodova.
- Kako definirati asinkronu akcijsku metodu u ASP.NET Core Web API-u i zašto je asinkroni pristup koristan za skalabilnost API-ja?
- Što je middleware u ASP.NET Core-u i koja mu je uloga? Kako middleware komponente sudjeluju u obradi HTTP zahtjeva?
- Navedi primjer ugrađenog middleware-a u ASP.NET Core pipelineu i ukratko objasni čemu služi (npr. middleware za posluživanje statičkih datoteka ili globalno rukovanje iznimkama).
- Kako se kreira i registrira vlastiti (custom) middleware u ASP.NET Core aplikaciji?
- Objasni pojam pipelinea obrade zahtjeva u ASP.NET Core-u. Kako redoslijed registriranih middleware komponenti utječe na ponašanje aplikacije?
- Kako omogućiti dokumentiranje i testiranje API-ja unutar same aplikacije? (Primjer: integracija alata Swagger/OpenAPI u ASP.NET Core Web API projekt.)
- Što je CORS (Cross-Origin Resource Sharing) i zašto ga je potrebno konfigurirati u Web API-ju? Kako omogućiti da web-aplikacija na drugoj domeni može slati zahtjeve tvom API-ju?
- Koja je razlika između autentifikacije i autorizacije u kontekstu web API-ja?
- Kako se u ASP.NET Core Web API-ju može implementirati autentifikacija korisnika? Opiši osnovne korake JWT autentifikacije (prijava korisnika, izdavanje tokena, provjera tokena pri svakom zahtjevu).
- Čemu služi atribut [Authorize] u ASP.NET Core Web API-u i kako ograničava pristup određenim endpointima?
- Kako ograničiti pristup određenoj API metodi samo korisnicima s određenom ulogom ili pravima? (Primjer: korištenje [Authorize(Roles="Admin")] ili definiranje prilagođene policy za autorizaciju.)

- Kako se može implementirati autorizacija na temelju politika (policy-based authorization) u ASP.NET Core-u i po čemu se razlikuje od autorizacije na temelju uloga?

Entity Framework Core i rad s bazom podataka

- Što je Entity Framework Core i koje probleme taj ORM rješava? Zašto bismo koristili EF Core umjesto da pišemo SQL upite ručno?
- Kako definirati klasu modela (entitet) za EF Core i kako se ta klasa preslikava u tablicu baze podataka?
- Kako se definira i konfigurira DbContext u EF Core-u? Objasni što predstavlja klasa DbContext unutar ASP.NET Core aplikacije.
- Kako u EF Core-u dodati novi zapis (entitet) u bazu i spremiti promjene? Opiši ukratko postupak dodavanja entiteta i poziva SaveChanges().
- Kako obrisati zapis iz baze pomoću EF Core-a? Što je potrebno napraviti s objektom entiteta prije spremanja promjena da bi se podatak uklonio?
- Kako se ažurira postojeći entitet i spremaju promjene u EF Core-u? Koje korake treba napraviti na objektu i DbContextu prije poziva SaveChanges()?
- Kako se izvode upiti u EF Core-u za dohvrat podataka iz baze? Opiši ukratko korištenje LINQ upita za filtriranje i projekciju podataka.
- Što je migracija u EF Core-u i čemu služi? Kako kreirati novu migraciju i primijeniti je na bazu podataka?
- Što su lazy loading, a što eager loading u EF Core-u? Kako eksplisitno učitati povezane entitete uz glavni entitet (npr. korištenjem metode .Include())?
- Što znači da EF Core prati entitete (change tracking) i kako upotreba metode AsNoTracking() utječe na performanse i praćenje promjena?
- Kako u EF Core-u modelirati odnose između entiteta (jedan-prema-jedan, jedan-prema-više, više-prema-više)? Ukratko opiši kako se mogu definirati strani ključevi ili koristiti Fluent API konfiguracija.
- Što je problem "N+1 upita" u kontekstu ORM-a i kako se može izbjegići u EF Core-u?
- Kako izvršiti raw SQL upit ili pohranjenu proceduru (stored procedure) kroz EF Core ako je potrebno? Koje metode EF Core nudi za direktno izvršavanje SQL naredbi?
- Navedi neke najbolje prakse za optimizaciju performansi upita u EF Core-u. Primjeri: projekcija samo potrebnih polja, korištenje AsNoTracking() za čitanje, dodavanje indeksa u bazu na često filtrirane stupce.
- Kako riješiti situaciju kada dva procesa ili korisnika istovremeno mijenjaju isti zapis u bazi preko EF Core-a? Objasni pojam optimistične konkurentnosti i kako ga EF Core podržava.

LINQ upiti i obrada podataka

- Što je LINQ (Language Integrated Query) u C# i koje prednosti donosi u radu s podacima i kolekcijama objekata?

- Kako filtrirati elemente kolekcije pomoću LINQ-a? (Primjer: uporaba metode Where za izdvajanje elemenata koji ispunjavaju određeni uvjet.)
- Kako transformirati (projicirati) kolekciju iz jednog oblika u drugi koristeći LINQ? (Primjer: uporaba metode Select za dobivanje kolekcije vrijednosti jednog svojstva objekta.)
- Kako ograničiti broj rezultata koje dobivamo LINQ upitom? (Primjer: korištenje metoda Skip i Take za paginaciju rezultata.)
- Kako sortirati rezultate LINQ upita? (Primjer: korištenje OrderBy i ThenBy za rastući poredak ili OrderByDescending za silazni poredak.)
- Koja su dva načina pisanja LINQ upita u C# i po čemu se razlikuju? (Objasni razliku između query sintakse i method sintakse.)
- Na koji način se mogu spojiti podaci iz dvije kolekcije pomoću LINQ-a? (Primjer: korištenje operacije join za spajanje dviju lista objekata na temelju zajedničkog ključa.)
- Kako grupirati kolekciju elemenata u LINQ-u? (Primjer: uporaba GroupBy za grupiranje objekata po nekom svojstvu i agregacija poput zbrajanja unutar svake grupe.)
- Kako primijeniti agregatne funkcije u LINQ upitima? (Primjer: korištenje metoda Sum, Average, Count za izračun vrijednosti iz kolekcije ili unutar grupe.)
- Kako provjeriti ispunjava li barem jedan element ili svi elementi kolekcije određeni uvjet? (Primjeri metoda Any i All u LINQ-u.)
- Što radi metoda SelectMany u LINQ-u i u kojem se slučaju koristi? Objasni kako ona može od više kolekcija napraviti jednu "spljoštenu" sekvencu.
- Što znači da LINQ koristi odgodeno izvršavanje (deferred execution)? Kada se LINQ upit zaista izvrši nad kolekcijom u memoriji ili nad bazom podataka?
- Koja je razlika između sučelja IEnumerable i IQueryable u kontekstu LINQ upita? Zašto je ta razlika bitna kod upita prema bazi podataka (LINQ to Entities)?
- Može li se bilo koja C# metoda ili funkcija koristiti unutar LINQ upita koji se izvršava nad bazom? Objasni zašto neki izrazi/funkcije nisu podržani u LINQ to Entities upitima.
- Navedi nekoliko načina kako poboljšati performanse LINQ upita. (Primjeri: filtriranje što ranije u upitu, pozivanje .ToList() u odgovarajuće vrijeme, izbjegavanje višestrukog iteriranja iste kolekcije.)

Dependency Injection i arhitektura aplikacije

- Što podrazumijeva pojам Dependency Injection (ubrizgavanje ovisnosti) i koji problem taj pristup rješava u razvoju softvera?
- Kako se primjenjuje Dependency Injection u ASP.NET Core aplikacijama? (Primjer: injektiranje servisa u kontroler putem konstruktora.)
- Koja je svrha datoteke appsettings.json u ASP.NET Core projektu? Kako se mogu koristiti različite konfiguracijske datoteke za različita okruženja (npr. Development vs. Production)?
- Što je IoC (Inversion of Control) container i kako ga ASP.NET Core koristi interno u sklopu implementacije DI-ja?
- Na koji način registriraš servise u ASP.NET Core aplikaciji za potrebe DI-ja? (Primjer: gdje se nalaze registracije i korištenje metoda poput AddTransient, AddScoped, AddSingleton.)

- Objasni razliku između Transient, Scoped i Singleton životnog vijeka (lifecycle) servisa u ASP.NET Core-u. Kada je prikladno koristiti koju od tih vrsta?
- Što znači da je kod “labavo spregnut” (loosely coupled)? Kako korištenje sučelja i DI-ja doprinosi labavoj sprezi komponenata u aplikaciji?
- Što je princip Dependency Inversion (slovo D u SOLID načelima) i kako se može primjeniti kroz korištenje DI-ja u praksi?
- Što je DTO (Data Transfer Object) i koja je njegova uloga u dizajnu Web API-ja? Zašto je često bolje vratiti DTO prema klijentu umjesto direktno entitet iz baze?
- Kako korištenje DI-ja i sučelja utječe na testiranje koda? Kako se može jednostavno testirati komponenta koja ima ovisnosti o drugim servisima?
- Kako pristupiti čitanju konfiguracijskih vrijednosti aplikacije (npr. konekcijskog stringa) u kodu koristeći DI? (Primjer: injektiranje sučelja IConfiguration ili korištenje Options patterna.)
- Koja je svrha filtera (npr. action filtera) u ASP.NET Core-u i kako se filteri razlikuju od middleware komponenata?
- Kako je implementiran sustav logiranja u ASP.NET Core-u? Kako se preko DI-ja može dobiti instanca ILogger i koristiti unutar vlastitih klasa za bilježenje logova?
- Kako se može strukturirati ASP.NET Core web-aplikacija u više slojeva i koje su odgovornosti svakog sloja? (Primjer: prezentacijski sloj – kontroleri, poslovni sloj – servisi, sloj pristupa podacima – repozitoriji.)
- Što je “repository” obrazac (Repository pattern) i zašto se koristi u slojevitim aplikacijama? Kako bi izgledala osnovna implementacija repositoryja u .NET backendu?

Osnove performansi i sigurnosti u backend aplikacijama

- Što je caching i kako može poboljšati performanse backend aplikacije? Koje vrste cachea postoje (npr. in-memory cache, distribuirani cache) i kada ih koristiti?
- Zašto je važno koristiti HTTPS (SSL/TLS) za komunikaciju s web API-jem? Kako osigurati da API prihvata samo sigurne (šifrirane) veze?
- Koje su prednosti asinkronog rukovanja zahtjevima (async/await) u ASP.NET Core Web API-u u odnosu na sinhrono? Kako asinkroni pristup utječe na broj istovremenih zahtjeva koje server može učinkovito obraditi?
- Što je connection pooling u kontekstu pristupa bazi podataka i kako utječe na brzinu obrade čestih zahtjeva prema bazi?
- Zašto je važno ograničiti količinu podataka koju API vraća odjednom (npr. uestvi paginaciju ili filtriranje rezultata)? Kako prevelik pojedinačni odziv može utjecati na performanse i resurse i klijenta i servera?
- Kako indeksiranje podataka u bazi (korištenje indeksa na stupcima) može utjecati na performanse upita koji se izvršavaju kroz vaš API? Zašto je bitno optimizirati bazu podataka za česte upite?
- Što je SQL injekcija i kako se aplikacija može zaštiti od SQL injekcijskih napada? (Primjeri obrane: parametarski upiti, korištenje ORM-a poput EF Core.)
- Kako sigurno pohraniti lozinke korisnika u bazi podataka? Zašto je hashiranje (sa soljenjem) lozinki bolje od čuvanja u čistom tekstu?

- Koje sigurnosne mjere treba primijeniti pri korištenju JWT tokena za autentifikaciju? (Primjeri: slanje tokena samo preko HTTPS-a, ograničavanje roka trajanja tokena, korištenje refresh tokena.)
- Što je CSRF (Cross-Site Request Forgery) napad i kako se može spriječiti? Trebaju li stateless REST API-ji (koji koriste JWT autentikaciju) brinuti o CSRF-u?
- Koja je razlika između vertikalne i horizontalne skalabilnosti aplikacije? Kako se svaki pristup može iskoristiti za povećanje mogućnosti poslužitelja da obradi veće opterećenje?
- Kako pristupiti identificiranju uskih grla (bottlenecks) u performansama aplikacije? Koje alate ili tehnike možeš koristiti za profiliranje i mjerenje performansi .NET aplikacije?
- Može li u .NET aplikaciji doći do curenja memorije (memory leak) unatoč automatskom sakupljanju smeća od strane GC-a? Kako prepoznati i spriječiti curenje memorije u aplikaciji koja dugo vremena radi pod opterećenjem?
- Kako spriječiti zlouporabu API-ja slanjem prevelikog broja zahtjeva u kratkom vremenu? Opiši koncept rate limitinga i kako pomaže u takvim situacijama.
- Gdje čuvati osjetljive konfiguracijske podatke aplikacije (poput konekcijskih stringova ili API ključeva) i kako ih zaštитiti? Zašto takve tajne ne treba držati u kodu ili javnom rezpositoriju i koje se metode koriste za sigurno upravljanje povjerljivim informacijama?

Hvala na čitanju
Sretno kodiranje

KRAJ.



