.NET WEB API

Tools

Kao IDE (Integrated development environment / Integrisano razvojno okruženje) ćemo koristiti Visual Studio.
Verzija .Net Framework-a koju ćemo koristiti je 7. Da bi mogli napraviti .NET 7 Web API projekat, moramo prvo instalirati .NET 7, ukoliko on nije već instaliran.
Link za download je: https://dotnet.microsoft.com/en-us/download/dotnet/7.0, naš operativni sistem je 64 bit-ni, tako da ćemo odabrati x64.
Da bi koristili .NET 7 moramo koristiti Visual Studio 2022 ili noviji

Napravi novi Web API projekat

1. Otvori VS 2022
2. Klikni na “Create a new project”
3. U pretrazi ukucaj “web api” i odaberi ASP .NET Core Web API iz liste (C#)
4. Dodijeli projektu ime i idi dalje
5. Odaberi .NET 7 kao Framework
6. Klikni na Creat

- **Program.cs** - je start-up klasa projekta. U njoj možemo da registrujemo servise, koje onda možemo globalno koristiti u čitavoj solution pomoću “***Dependency Injection***”. app.Use-itd metode su tu da dodamo funkcije u ***request pipeline***.
- **Swagger** - Od .NET 6 Swagger je po default konfigurisan za solution. Swagger kreira web stranicu koja se otvori kada pokrenemo solution, pomoću koje možemo da testiramo naše ***endpoints***.
- **Appsettings.json** - Je JSON file u koji možemo da spašavamo naše konfiguracije.

Prvi API poziv

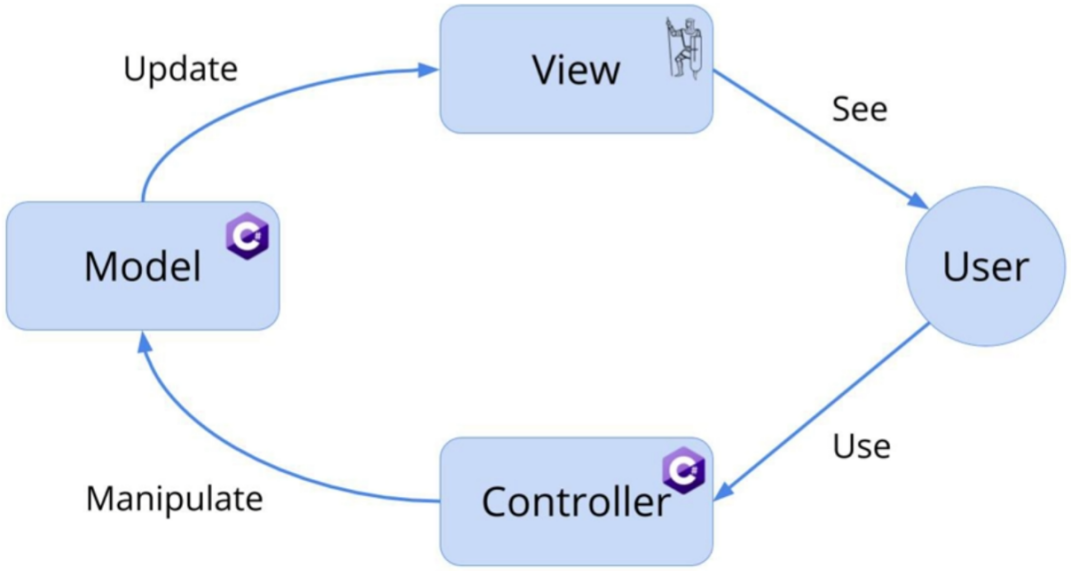
- **Route** je putanja koja će usmjeriti nadolazeći request/upit na endpoint/funkciju, u kontroleru, kojoj smo je pridružili
- **Swagger Schema** sadrži listu definicija klasa koje se koriste u kontrolerima
- **Parametri** su podaci koje možemo slati u upitu isto kao kod običnih funkcija
- **Request** je upit koji se šalje na endpoint
- **Response** je odgovor koji se dobije kao rezultat upita

GIT Repository

1. Klikni na View
2. Klikni na git changes
3. U GIT changes prozoru klikni na Create Git Repository
4. Unesi Repository Name
5. Klikni Create and Push

MVC Pattern (Model-View-Controller)

Kod MVC uzorka, Model predstavlja klase, View predstavlja UI (user interface/Frontend) i Controller predstavlja logiku koja manipulira podacima (funkcije, servisi itd.). U Backendu ćemo samo raditi Model i Controller.



Novi Model (Character)

U ovom kursu ćemo napraviti RPG igricu, gdje možemo napraviti nove likove i pustit ih da se bore.
Prvo napravi folder koji se zove Models, u koji ćemo spašavati sve modele koje budemo pravili.

Napravi klasu Character sa sljedećim poljima:

- *Int Id*
- *String Name sa default vrijednost "Frodo"*
- *Int HitPoints (Koliko mu napad oduzima poena) = 100*
- *Int Strength (Snaga) = 10*
- *Int Defense (Odbrana) = 10*
- *Int Intelligence (inteligencija, za magiju) = 10-*

- *RPGEnum Class = Knight*

Napravi Enum RPGClass: Knight, Mage, Cleric i dodaj ga kao polje u klasu Character

Prvi Kontroler

Kontroler (Controller) je klasa sa kojom grupišemo funkcije koje se zovu **Action Methods**.
Action Methods su funkcije koje obrađuju zahtjeve (**requests**) koje dolaze sa Frontend-a.
Sve kontrolere koje budemo pravili, moramo praviti u folder-u Controllers.
Svaki kontroler mora imati sufix Controller u imenu, npr. UserController, OrderController itd.
Da bi napravili novi kontroler, kliknut ćemo desnu tipku na folder Controllers i odabrati opciju Add > Controller > API Controller Empty.

[ApiController] je atribut koji govori compiler-u da klasa koja se nalazi ispod njega je Kontroler, te dodijeliti toj klasi određene funkcionalnosti koje su joj potrebne da bi funkcionisala kao Kontroler (Routing, Validacija, Binding itd.).

[Route] je atribut sa kojim definišemo putanju do tog kontrolera. Ako se kontroler zove **OrderController** onda će po default-u putanja biti **/api/Order**.

ControllerBase je bazna klasa koju treba svaki kontroler da nasljeđuje

Napravi novi kontroler sa imenom CharacterController i dodaj novu Action metodu u njega koja će samo vratiti objekat Character nazad.

Nova Action metoda mora biti označena sa HttpGet atributom i vratiti tip ActionResult<Character>.

[HttpGet] je atribut koji označava da action metoda ispod njega prima samo HTTP Get zahtijeve. Kasnije ćemo ući dublje u tu temu.

ActionResult je klasa koja se koristi kao povratni tip, koji pored podataka može da vrati još dodatne informacije o tome da li je zahtjev uspješno obrađen ili je imao grešaka itd. U obliku **Status code-a**

Status codes

1xxs – Informativni odgovori

2xxs – Uspjeh! Zahtjev je uspješno završen i server je pretraživaču dao očekivani odgovor

3xxs – Preusmjeravanje: Preusmjereni ste negdje drugdje. Zahtjev je primljen, ali postoji neka vrsta preusmjeravanja.

4xxs – Greške klijenta: Stranica nije pronađena. Nije moguće pristupiti web stranici ili stranici.

5xxs – Greške servera: greška. Klijent je dao ispravan zahtjev, ali server nije uspio dovršiti zahtjev.

Popularni Status Codes:

- **200 (Ok) -** Request je prošao uspješno
- **404 (Not found) -** Stranica nije pronađena (Frontend)
- **410 (Gone) -** Stranica više ne postoji
- **500 (Internal server error) -** Najčešći server error code, ukazuje da je request ispravan ali se greška desila na backendu
- **503 (Service unavailable) -** Server/Backend nike dostupan

Parametar (Route parameter)

Kao kod obične funkcije, možemo slati parametre i action methods. Jedan od načina slanja parametara u action method je preko njegove putanje (route). Ova vrsta parametara se zove **route parameter**.

```
[HttpGet("GetSingle/{id}")]
Public ActionResult<Character>(int id) { … }
```

Da bi action method mogla da primi route parameter, moramo u putanji navesti da action method može da primi parameter. Ime parametra u putanji mora da se nalazi u vitičastim zagradama i mora biti podijeljeno od ostatka putanje sa kosom crtom. ***/{id}***
Drugi uslov je da moramo definisati parametar sa istim imenom kao u ruti u zagradama action methode, isto kao i kod obične funkcije.
U rutu možemo staviti više parametara, ali redoslijed parametara u glavi action methode mora biti isti kao u putanji.

HTTP Metode

Hypertext Transfer Protocol (HTTP) je dizajniran da omogući komunikaciju između klijenata (frontend) i servera (backend).

HTTP radi kao protokol zahtjev-odgovor između klijenta i servera.

Primjer: Klijent (pretraživač) šalje HTTP zahtjev serveru; tada server vraća odgovor klijentu. Odgovor sadrži informacije o statusu zahtjeva, a može sadržavati i tražene podatke.

Najčešće korištene HTTP metode su:

- *GET (Read) - se koristi za traženje podataka sa servera. Npr: Pošalji mi listu svih korisnika*
- *POST (Create) - se koristi za slanje podataka na server, da bi spasili nove podatke. Npr. Kada napravimo novi račun na stranici, ona ga šalje na server sa POST zahtijevom da bi ga spasila u DB*
- *PUT (Update) - se također koristi za slanje podataka na server, ali u svrhu da editujemo postojeće podatke. Npr. Kada promijeniš opis slike na Instagram, on šalje PUT zahtjev na server da bi promijenio opis postojeće slike*
- *DELETE (Delete) - se koristi da bi izbrisali podatke. Npr. Kada izbrišemo email, šalje se Delete zahtjev na server sa Id-om maila da bi ga server izbrisao iz DB*

CRUD = Create (POST) - Read (GET) - Update (PUT) - Delete (DELETE)

JSON

JavaScript Object Notation (JSON) je standardni format zasnovan na tekstu za predstavljanje strukturiranih podataka zasnovanih na sintaksi JavaScript objekata. Obično se koristi za prijenos podataka u web aplikacijama (npr. slanje nekih podataka sa servera klijentu, kako bi se mogli prikazati na web stranici, ili obrnuto).

```
{
        Id: 1,    ⇐ Ovo je int
        FirstName: “Merjem”,    ⇐ Ovo je string
        LastName: “Zalihic”,
        Order: {          ⇐ Ovo je objekat
                Id: 1,
                Total: 215.12,
                Date: 2023-01-15
        },
        Predmeti: [ “PR1”, “PR2”, “Baze podataka” ]    ⇐ Ovo je lista stringova
}
```

*JSON članovi se sastoje uvijek iz dva dijela **Ključ/Ime : Vrijednost***
Vrijednost može da bude primitivan tip kao: int, string, double itd. Ali može da bude i tipa objekat i lista.
*Ako je vrijednost tipa objekat, onda moramo staviti vitičaste zagrade i u njih pisati članove top objekta, opet u formatu **ključ:vrijednost.***
Ako je vrijednost tipa lista, onda moramo staviti kockaste zagrade i u njih onda članove liste.

Parametar (Body parameter)

Ukoliko želimo da šaljemo objekte na server, onda ih moramo slati preko request body-a. Objekti se ne mogu slati preko ***route parametra.***
Kada želimo da action method prima body parametar, dovoljno je da definišemo tip objekta i njegovo ime u glavi action methode, kao kod običnih funkcija.
Dodatno možemo ukrasiti parametar sa atributom ***[FROMBODY]****, ali ne moramo.*
!! Body parametri se samo mogu slati sa POST i PUT metodama.

```
[HttpPost("CreateCharacter")]
Public ActionResult<Character>(Character newCharacter) { … }
ILI
[HttpPost("CreateCharacter")]
Public ActionResult<Character>([FromBody] Character newCharacter) { … }  <= Opcionalno kod .NET 7
```

Servisi i dependency injection

Kontroleri i njihove action metode bi trebale samo primiti request i vratiti response, iz tog razloga sva logika obrade podataka se po uzorku prebacuje na klase koje se zovu servisi. Servisi su jednostavne klase koje sadrze funkcije, koje trebaju da prime zahtjev od kontrolera i vrate njemu odgovarajući rezultat.
Servisi se pozivaju u kontroleru pomoću ***dependency injection.***
Dependency injection *je uzorak koji omogućava da ubrizgamo klase u druge klase koje ovise o njima (npr. Da ubrizgamo servis u kontroler, posto kontroler ovisi o servisu, pošto mu on mora obraditi i vratiti rezultat).*
Kako napraviti servis i ubrizgati ga u kontroler pomoću dependency injection:

1. *Napravi interface koji definiše sve funkcije koje servis treba da implementira*
2. *Napravi novu klasu (servis) koja implementira interface. Servis i interface obično dijele ime. Ako se servis zove UserService onda će se interface zvati IUserServise. Ispred imena interface-a uvijek ide slovo "I"*
3. *Registruj servis u Program.cs*
   builder.Services.AddTransient<ICharacterService, CharacterService>()
4. *U kontroleru napravi privatni atribut/varijablu tipa interface-a i napravi konstruktor. Konstruktor treba da prima interface kao parametar, pošto ga tako ubrizgavamo servis u kontroler pomoću* ***DI****. Zatim spasi parametar u atribut što si napravila*

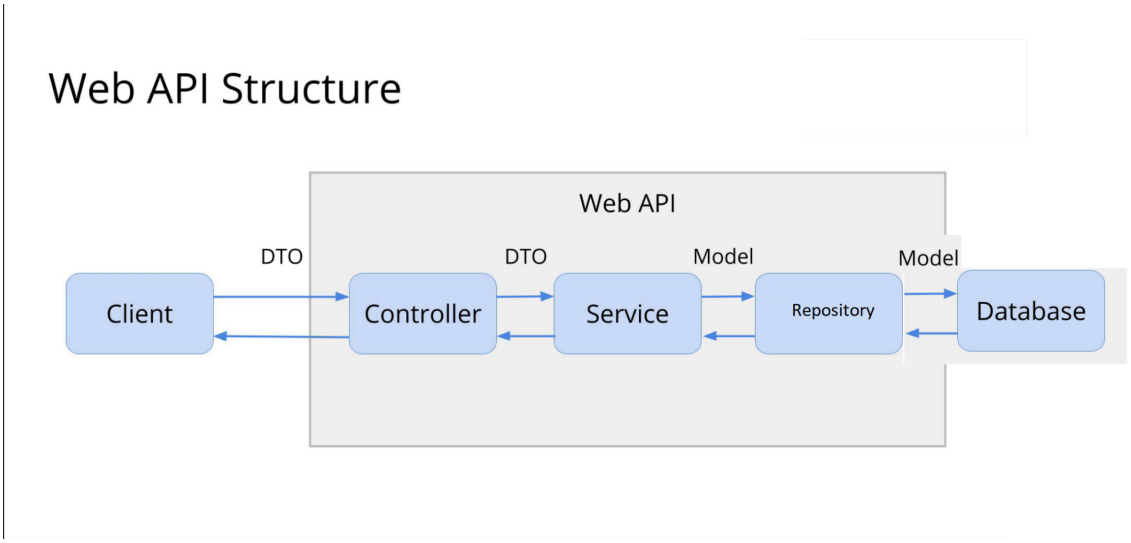
```
private ICharacterService _characterService;

public CharacterController(ICharacterService characterService)
{
   _characterService = characterService;
}
```

DTO i AutoMapper

DTO (data transfer object) je objekat koji prenosi podatke između dijelova aplikacije, kao npr. sa Frontend-a ka Backend-u ili obrnuto. Modeli, za koje smo rekli da preslikavaju tabele iz DB se ne bi trebali nikada slati Frontend-u, iz tog razloga koristimo DTOs da bi definisali samo one podatke koje su potrebne Frontendu i koje on smije da vidi.

DTO definišemo kao klasu, isto kao i Model, jedina razlika je što DTO možemo proširiti sa dodatnim podacima ili sakriti neke podatke koje Frontend ne smije da vidi, kao npr. Lozinka.

Podaci koji dolaze iz DB prema Backend-u se prvenstveno spašavaju u Modelu, pošto Model preslikava tabelu iz DB 1:1. Zatim Modele trebamo konvertovati u DTO prije nego ih pošaljemo na Frontend.



Jedan od načina da to postignemo jeste da koristimo biblioteku AutoMapper.

AutoMapper je biblioteka sa kojom možemo preslikavati (mappati) vrijednosti iz jednog objekta u drugi. Ti objekti su još označeni kao ***izvor,*** *iz kojeg se vrijednosti kopiraju, i* ***destinacija*** *u koji se vrijednosti kopiraju.*

Vrijednosti će se samo preslikavati između atributa objekata koji imaju isto ime i tip.

Npr. Ukoliko izvor ima atribut ***public int YearOfBirth { get; set; }*** *a destinacija nema atribut sa istim imenom i tipom, ili obrnuto, onda te taj atribut neće preslikati u destinaciju.*

Da bi AutoMapper znao koje klase može da preslikava, moramo napraviti Profile klasu u kojoj to definišemo, npr:

CreateMap<User, UserDto>().ReverseMap();

*Nakon što smo definisali Profile klasu, trebamo je registrovati u **Program.cs:***

```
builder.Services.AddAutoMapper(x =>
{
   x.AddProfile<DomainProfile>();
});
```

*Zatim možemo AutoMapper koristiti u klasama pomoću **dependency injection** preko njegovog interface-a:*

```
private readonly IMapper _mapper;

public UserService(IMapper mapper)
      {
         _mapper = mapper;
      }
```

Da bi preslikali jedan objekat u drugi, koristimo funkciju Map.

U sljedećem primjeru preslikavamo ProductDTO u Product Model. Između <...> stavljamo u koji tip (klasu) preslikavamo a zatim kao parametar šaljemo onaj objekat koji preslikavamo.

```
Product newProduct = _mapper.Map<Product>(productDto);
```

Entity Framework

"Entity Framework je objektno-relacijski maper (ORM) koji omogućava .NET programerima da rade s bazom podataka koristeći .NET objekte. To eliminira potrebu za većinom koda za pristup podacima koji programeri obično trebaju da napišu."

EF je biblioteka koja nam omogućava da komuniciramo sa bazom podataka, koristeći Klase koje još nazivamo Modele. Ideja je da čitav pristup podacima u Bazi Podataka bude moguć pomoću samo C#-a bez da moramo koristiti SQL. Funkcioniše tako što napravimo klase, zvane Modeli, koje preslikavaju tabele iz baze podataka, pri čemu atributi klase preslikavaju kolone Tabele iz Baze Podataka.

Također nam omogućava da radimo Code First, što znači da možemo napraviti prvo Modele pa onda narediti EF-u da generiše Tabele u Bazi Podataka na osnovu tih Modela.

*Jedna od poslijedica tog pristupa je da, kada god **dodamo novi Model, izbrišemo Model ili promijenimo neki Model**, moramo napraviti Migraciju, koja će te naše promijene u Kodu preslikati na Bazu Podataka.*

Instalacija

Da bi koristili EF, moramo prvo instalirati sve potrebne biblioke pomoću NuGet Manager, a te biblioteke su:

- *Microsoft.EntityFrameworkCore*
- *Microsoft.EntityFrameworkCore.Design*
- *Microsoft.EntityFrameworkCore.SqlServer*
- *Microsoft.EntityFrameworkCore.Tools*

Konfiguracija

*Nakon što smo instalirali sve potrebne biblioteke, moramo konfigurisati EF tako što ćemo napraviti **Data Context.***

***Data Context** napravimo kao običnu klasu, a zatim naslijedimo **DbContext** klasu iz Microsoft.EntityFrameworkCore biblioteke, te u konstrukturu proslijedimo DbContextOptions u baznu klasu:*

```
public class DataContext : DbContext
{
    public DataContext(DbContextOptions options) : base(options) { }
    …
}
```

*Zatim u istoj klasi za svaki Model napravimo **DbSet**. DbSet govori EF-u koje tabele treba da napravi i održava u bazi podataka:*

```
public DbSet<Character> Characters { get; set; }
public DbSet<Product> Products { get; set; }
public DbSet<Order> Orders { get; set; }
public DbSet<User> Users { get; set; }
```

*Da bi EF znao sa kojim DB Serverom da komunicira i koju bazu sa tog servera da koristimo, moramo mu proslijediti **Connection String**.*
*Taj Connection String možemo spasiti u **appSettings.json**, odakle ga onda možemo kasnije isčitati:*

U appSettings.json dodamo slijedeći unos:

```
"ConnectionStrings": {
  "Development": "Server=localhost;Database=RPGGame;Trusted_Connection=true;TrustServerCertificate=true;"
},
```

- *Server označava na kojem Serveru se nalazi DB. Ukoliko napišemo localhost ili stavimo samo tačku ".", to znači da je računar na kojem radimo trenutno Server.*
- *Database: je ime DB koju treba da koristi*

- *Trusted_connection: znači da mu ne treba username i password da bi se spojio na Server*

Nakon što smo to završili, moramo registrovati EF u Program.cs, isto kao i sve service i AutoMapper:

```
builder.Services.AddDbContext<DataContext>(x =>
{
    x.UseSqlServer(builder.Configuration.GetConnectionString("Development"));
});
```

builder.Configuration.GetConnectionString("Development") => *Ovako isčitavamo connection string iz appSettings.json, koji ima ključ "Developer".*

Nakon što smo konfigurisali sve to, moramo napraviti prvu migraciju, tako što ćemo otvoriti NuGet Console, i izvršiti sljedeće dvije komande:

- *Add-Migration NekiNazivMigracije*
- *Update-Database*

Prva se koristi za kreiranje migracije, a druga da izvršimo tu migraciju u DB.
Ove dvije komande moramo izvršiti kada god:

- *Dodamo novi Model i definišemo za njega DbSet*
- *Izbrišemo DbSet i njegov Model*
- *Promijenimo neki Model za koji postoji DbSet. Npr. dodamo ili izbrišemo attribute*

Dodatna Konfiguracija

Da bi mogli koristiti naš DataContext i njegove funkcije u ostatku našeg Projekta, moramo napraviti interface za naš Data Context, te registrovati Data Context u Program.cs isto kao i sve service.

Interface nam može izgledati ovako:

```
public interface IDataContext
{
   void Seed();
    int SaveChanges();
    DatabaseFacade Database { get; }
}
```

Zatim naš Data Context mora implementirati ovaj interface:

```
public class DataContext : DbContext, IDataContext
```

Seed funkciju koju smo definisali u interface-u moramo sada implementirati u Data Context, možemo je ostaviti praznu za početak:

```
public void Seed() {
};
```

Kada smo to sve završili, možemo Data Context registrovati u Program.cs kao i sve ostale servise kao Transient:

```
builder.Services.AddTransient<IDataContext, DataContext>();
```

Sada možemo koristiti DataContext u svim klasama pomoću Dependency Injection, isto kao i servise.

Repository

Repository je uzorak, kao i Servisi, s kojim želimo da odvojimo čitavu logiku pristupa Bazi Podataka u zasebne klase, zvane Repositories.

Repository se nalazi između Servisa i Baze Podataka i radi isključivo sa Modelima a ne sa DTOs.

Zadatak Repository-a je da šalje komande na Bazu Podataka i na taj način čita podatke iz nje ili unosi nove podatke u nju.

U slučaju Create i Update npr. bi Repository primio Model od servisa i onda ga dodao kao novi zapis u DB ukoliko se radi o Create ili prepisao postojeći zapis u DB ukoliko se radi o Update.

Kao povratni tip Repository uvijek vraća Model.

Da bi Repository mogao komunicirati sa DB, koristit će Entity Framework, odnosno DataContext koji smo konfigurisali (pogledaj poglavlje Entity Framework).

Repository se pravi isto kao i Servis:

1. *Napravi interface koji definiše sve funkcije koje će imati Repository*
2. *Napravi Repository klasu i implementiraj interface*
3. *Registruj Repository kao Transient u Program.cs*

Da bi mogli slati komande DB-u u Repository, ubrizgat ćemo u njega naš DataContext.

```
    private readonly IDataContext Context;

    public RepositoryBase(IDataContext dataContext)
    {
        Context = dataContext;
    }
```

*Kada god koristimo Data Context, moramo mu uvijek reći sa kojom tabelom želimo da radimo. To ćemo postići pomoću funkcije **Set** u DataContext koja je generična. Moramo staviti uvijek ime Modela sa kojim želimo da radimo između < > kod funkcije **Set.***

Npr. Ako želimo da radimo sa tabelom User, pristup bi ovako izgledao:

```
Context.Set<User>()
```

*Nakon Set, možemo koristiti sve iste funkcije koje ima i **List,** kao npr:*

- *FirstOrDefault => Da dohvati samo jednog na osnovu lambda uslova koji napišemo*
- *Add => Dodaj novi objekat u tabelu*
- *Remove => Izbriši zapis iz tabele*
- *ToList => Vrati sve zapise iz te tabele*

***Važno**, nakon što uradimo neku promjenu u tabeli, kao npr Add ili Remove, moramo pozvati funkciju SaveChanges() iz Data Context. Tek nakon SaveChanges() će se promijene i oslikati u DB.*

Npr:

```
Context.Set<User>().Add(userModel);
Context.SaveChanges();
```

Pošto se neke funkcije stalno ponavljaju u Repository, kao što su CRUD operacije, pošto skoro uvijek moramo imati CRUD funkcije za svaku tabelu u DB, možemo napraviti jedan Bazni Repository koji će biti generičan i imati sve te CRUD funkcije u sebi. Na taj način samo moramo tom Base Repository reći sa kojom tabelom da radi i ne moramo za svaku tabelu praviti novi Repository sa istim CRUD funkcijama.

Prvi korak, napravimo novi Repository koji je generičan i koji samo prima klase koje nasljeđuju naš bazni Model, u kojem smo napisali atribut Id i kojeg nasljeđuje svaki naš Model. Na onaj način ćemo osigurati da će svaki objekat koji pošaljemo u ovaj Repository imati atribut Id.

```
public class RepositoryBase<T> : IRepositoryBase<T> where T : BaseModel
```

Drugi korak, ubrizgamo Data Context pomoću Dependency Injection:

```
private readonly IDataContext Context;

public RepositoryBase(IDataContext dataContext)
{
    Context = dataContext;
}
```

Treći korak, napravimo sve CRUD funkcije.

Npr. funkcija da izvućemo Zapis iz Baze preko njegovog ID-a bi izgledalo ovako:

```
public virtual T GetById (int id)
{
  return Context.Set<T>().FirstOrDefault(x => x.Id == id)
}
```

Virtual stavljamo, da bi mogli prepisati ovu funkciju po potrebi, ako praviti neki drugi Repository koji nasljeđuje ovaj BaseRepository.

Kako smo implementirali Create, Update, Delete, GetAll itd. Možeš viditi u projektu RPGGame > Repositories > RepositoryBase > RepositoryBase.cs

Četvrti korak, pozovi Repository Base u nekom Servisu.
Da bi mogli koristiti Repository Base, moramo ga ubrizgati pomoću Dependency Injection u Servis i reći mu sa kojim Model radi:

```
private readonly IRepositoryBase<User> _repository;
```

Poziv funkcije GetById za tabelu User bi izgledao ovako:

```
User user = _repository.GetById(id);
```

Autentikacija i .NET Cookie

Autentikacija je čin dokazivanja identiteta. U svijetu aplikacija autentikacija se obično vrši tako što se korisnik loguje sa svojim username i password. Autentikacija je važna da bi zaštitili naš API od poziva koji dolaze od osoba ili robota koji nisu logirani u našu aplikaciju.

Sam čin autentikacije pomoću username i password je relativno jednostavan:

1. *Korisnik unese username i password na Frontendu*
2. *Frontend zatim pošalje username i password na API*
3. *U API mi provjerimo da li postoji korisnik sa tim username u DB i da li je password koji je unjeo korektan*

Međutim, ukoliko API ne upamti da se ovaj korisnik logirao, on bi se morao logirati svaki put prije nego što pošalje zahtjev na API.

Iz tog razloga u igru dolazi Cookie.
Nakon što utvrdimo da su username i password korektni, trebamo da napravimo jedan Cookie za tog korisnika.

U Cookie spasimo sve relevantne podatke o korisniku, kao npr. Ime, Prezime, ID, Username.
Nakon što napravimo Cookie, API ga automatski šalje nazad Frontendu, nakon čega ga Browser automatski spasi u svoju memoriju.
Od tog trenutka će Browser sa svakim Request-om na naš API poslati i taj Cookie, tako da je Cookie zapravo kao potpis od Korisnika i API zna za svaki request ko ga je poslao, pomoću Cookie-a.

Cookie prvo moramo omogućiti u našoj aplikaciji. To radimo u Program.cs tako na slijedeći način:

```
builder.Services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(opts =>
    {
        opts.ExpireTimeSpan = TimeSpan.FromDays(7);
    });
```

Opcija ExpireTimeSpan nam omogućava da odreimo koliko dugo će Cookie biti validan, nakon što istekne taj period, korisnik će se morati ponobo logirati.

Zatim moramo dodati i sljedeće dvije linije koda, ali nakon linije koda

***var app = builder.Build();**:*

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

Funkcija za kreiranje Cookie-a bi izgledala onda ovako:

```
        private async Task CreateCookie(User user)
        {
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.Email, user.Email),
                new Claim(ClaimTypes.GivenName, user.FirstName),
                new Claim(ClaimTypes.Surname, user.LastName),
                new Claim(ClaimTypes.NameIdentifier, user.Id.ToString())
            };

            await HttpContextAccessor.HttpContext.SignInAsync(
                        CookieAuthenticationDefaults.AuthenticationScheme,
                        new ClaimsPrincipal(new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme)));
        }
```