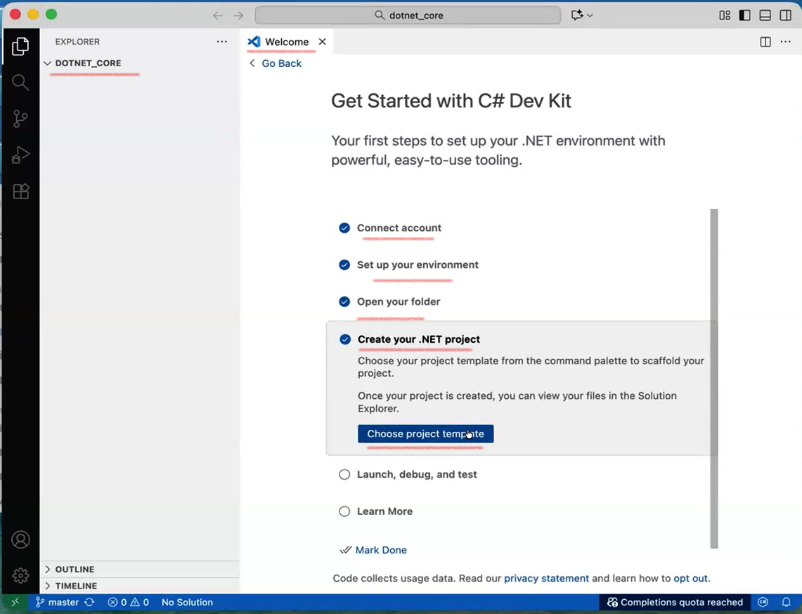
DOTNET CORE WepApi

# Days\_78 dotnet\_core proje kurulumu \*\*workspace/dotenet\_core/restApi---> 12/10/2025



**Yeni proje oluşturma adımları:**

* vs code -> welcome sayfasından--> **get started with c# dev kit**
* **open folder -->** klasör seçimi (dotnet\_core ana dizin olarak

belirle).

* **ctrl + shift +p --> ASP.NET Core WebAPI’si** proje şablonunu seç
* **RestApi** diye ad verdik
* Formatı --> **.sln** seçtik
* Ve create✅✅

**Gerekli kurulum ve bağımlılıklar: (**terminalden sıra ile yükleyebilirsin)

**SQLite**

<https://github.com/sqlitebrowser/sqlitebrowser/releases/download/v3.13.1/DB.Browser.for.SQLite-v3.13.1-win64.msi>

dotnet add package Microsoft.EntityFrameworkCore.Sqlite

dotnet add package Microsoft.EntityFrameworkCore.Design

dotnet add package Microsoft.EntityFrameworkCore.Tools

**Password Şifreleme için**

dotnet add package BCrypt.Net-Next

**JWT**

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

**Package GUI**

<https://marketplace.visualstudio.com/items?itemName=aliasadidev.nugetpackagemanagergui>

**Tablo- Migrations Oluşturma**

dotnet ef migrations add AppTables

dotnet ef database update (\*\*1.20dk da gql lite new database kurulumu anlatıldı)

**📘 .NET Core Nedir?**

**.NET Core**, Microsoft tarafından geliştirilen, açık kaynaklı (**open-source**) ve çoklu platform (**cross-platform**) desteğine sahip modern bir uygulama geliştirme çatısıdır (framework). 2016 yılında yayınlanmış ve klasik **.NET Framework**’ün kısıtlamalarını ortadan kaldırmak için tasarlanmıştır. 2020 yılından itibaren “.NET 5, .NET 6, .NET 7, .NET 8” adlarıyla devam etmektedir — yani günümüzde kullanılan haliyle artık **“.NET”** diye anılır.

**🔍 1. .NET Core’un Ortaya Çıkış Amacı** Klasik **.NET Framework** sadece **Windows** üzerinde çalışabiliyordu. Ancak yazılım dünyası zamanla **Linux, macOS, Docker, bulut sistemleri (cloud)** gibi farklı ortamlara yöneldi.

Microsoft bu değişimi yakalayabilmek için:

* Platform bağımlılığını kaldırmak,
* Performansı artırmak,
* Açık kaynaklı bir ekosistem oluşturmak  
  amacıyla **.NET Core’u** geliştirdi.

**⚙️ 2. .NET Core’un Temel İşleyişi**

Bir .NET Core uygulaması genel olarak şu şekilde işler:

1. **Kaynak Kod (C# / F# / VB.NET)**  
   → Geliştirici tarafından yazılan kod (.cs dosyaları).
2. **Derleyici (Roslyn)**  
   → C# kodunu **IL (Intermediate Language)** denen ara dile dönüştürür.
3. **.NET Core Runtime (CoreCLR)**  
   → IL kodunu **JIT (Just-In-Time)** derleyicisiyle işletim sistemine uygun **makine koduna** çevirir.
4. **.NET Core Libraries (CoreFX)**  
   → Uygulamanın çalışması için gerekli sistem fonksiyonlarını sağlar (dosya okuma, HTTP, matematik, koleksiyonlar, vs).
5. **Platform Katmanı**  
   → Uygulama artık işletim sisteminde (Windows, Linux, macOS) çalışır.

📊 **Kısaca:** C# kodu → IL → JIT derlemesi → Makine kodu → Çalışan uygulama

**🧩 3. .NET Core’un Temel Özellikleri**

| **Özellik** | **Açıklama** |
| --- | --- |
| **Cross-platform** | Windows, macOS ve Linux üzerinde çalışabilir. |
| **Open Source** | GitHub’da kaynak kodu herkese açıktır. |
| **Yüksek Performans** | Özellikle ASP.NET Core ile web uygulamalarında ciddi hız artışı sağlar. |
| **Modüler Mimari** | Gereken kütüphaneler yüklenir, gereksiz dosyalar kullanılmaz. |
| **CLI Desteği (Command Line Interface)** | Terminal üzerinden proje oluşturma, derleme, çalıştırma gibi işlemler yapılabilir. |
| **Microservices Desteği** | Docker ve Kubernetes ile kolay entegre olur. |
| **Cloud Entegrasyonu** | Azure, AWS gibi bulut sistemlerinde rahat çalışır. |
| **Backward Compatibility** | Eski .NET Framework kütüphanelerinin çoğunu destekler. |

**🧱 4. .NET Core Mimarisi**

**Uygulama Katmanı (Presentation Layer)**

**↓**

**İş Katmanı (Business Logic Layer)**

**↓**

**Veri Erişim Katmanı (Data Access Layer)**

**↓**

**Veritabanı (SQL Server, PostgreSQL, MongoDB, vb.)**

Bu yapı **katmanlı mimari** olarak bilinir ve kodun düzenli, test edilebilir, genişletilebilir olmasını sağlar.

**🧰 5. .NET Core’un Günlük Hayattaki Kullanımı**

.NET Core, bugün birçok gerçek dünya uygulamasında aktif olarak kullanılıyor. İşte bazı örnekler:

| **Kullanım Alanı** | **Gerçek Hayattaki Örnek** |
| --- | --- |
| 🏦 **Finans Sistemleri** | Banka API’leri, ödeme sistemleri, POS altyapıları. |
| 🏥 **Sağlık Sektörü** | Hastane bilgi sistemleri, hasta takip yazılımları. |
| 🏭 **Savunma ve Endüstri** | Aselsan, Havelsan gibi kurumlarda kontrol ve veri analiz sistemleri. |
| 🛒 **E-Ticaret** | Web siteleri, stok ve fatura yönetimi sistemleri. |
| 🧑‍💻 **Kurumsal Uygulamalar** | İnsan kaynakları (HR), ERP, CRM sistemleri. |
| ☁️ **Bulut Tabanlı Sistemler** | Azure üzerinde çalışan mikroservis tabanlı uygulamalar. |
| 🔐 **Kimlik Doğrulama Sistemleri** | Token (JWT), OAuth tabanlı login/register altyapıları. |

**🚀 6. .NET Core’un Modern Yazılım Dünyasındaki Rolü**

**🌐 Web Geliştirme**

ASP.NET Core sayesinde dinamik web siteleri, REST API’ler ve servis tabanlı sistemler geliştirilir. Performans olarak Node.js ve Django gibi framework’lerle rekabet eder.

**🧮 Mikroservis Mimarisi**

.NET Core’un hafif yapısı ve Docker desteği sayesinde mikroservis tabanlı sistemler kolaylıkla inşa edilir.

**☁️ Bulut Teknolojileri**

Microsoft Azure, AWS Lambda gibi bulut sistemlerine kolayca deploy edilir.

**📱 Mobil ve Masaüstü**

.NET MAUI ile mobil (Android, iOS) ve masaüstü (Windows, macOS) uygulamalar tek kod tabanıyla yapılabilir.

**🔄 7. .NET Core Ekosistemi**

| **Bileşen** | **Açıklama** |
| --- | --- |
| **ASP.NET Core** | Web uygulamaları ve API geliştirme için. |
| **Entity Framework Core (EF Core)** | Veritabanı işlemleri (ORM) için. |
| **.NET CLI** | Komut satırı araçları. |
| **.NET SDK** | Geliştirme araç seti (derleyici + kütüphaneler). |
| **.NET Runtime** | Uygulamanın çalışmasını sağlayan ortam. |
| **NuGet** | Paket yönetim sistemi. |

**📈 8. .NET Core’un Geleceği**

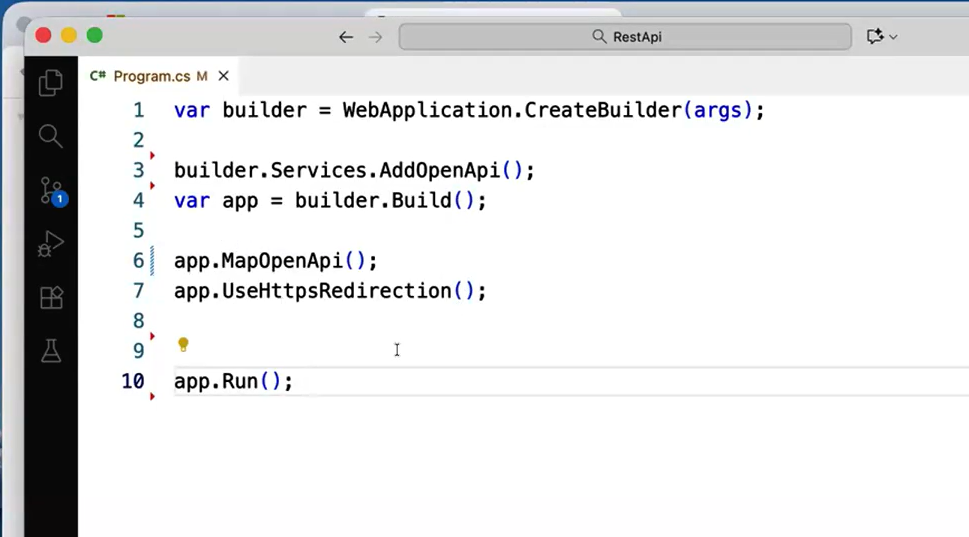
Microsoft, artık tüm yatırımlarını **.NET 8 ve sonrası** sürümler üzerine yapıyor.  
Yeni sürümler:

* Daha **yüksek performans**,
* **AI (Yapay zekâ)** entegrasyonları,
* **Minimal API**,
* **Native AOT (ahead-of-time compilation)** desteği gibi yenilikler içeriyor.

Bu da .NET Core’u uzun vadede yazılım dünyasının en güçlü platformlarından biri haline getiriyor.

**🧠 Özetle**

| **Özellik** | **.NET Core’un Katkısı** |
| --- | --- |
| 🚀 Performans | Çok hızlı çalışma süresi ve düşük bellek kullanımı |
| 🌍 Taşınabilirlik | Her işletim sisteminde çalışma |
| 🔧 Esneklik | Microservice, API, web, masaüstü, mobil hepsi tek çatı altında |
| 🧩 Modülerlik | Gereken bileşenleri yükleme imkânı |
| 🔒 Güvenlik | Modern kimlik doğrulama ve güvenlik sistemleri |
| 💡 Modernlik | Yeni nesil C# sürümleriyle entegre yapı |



Program.cs içinde olması gereken temel kodları bırakıp gerisini temizliyoruz. Ve amaca uygun tasarlamaya başlıyoruz…

**📘 Program.cs Dosyasının Amacı**

Bu dosya, uygulamanın çalışmaya başladığı **ana dosyadır**. Uygulamanın servislerini tanımlar, yapılandırmaları (config) yükler ve HTTP pipeline’ını (middleware zinciri) oluşturur. Yani özetle:

* Veritabanı bağlantısı,
* Controller tanımları,
* Swagger (OpenAPI)
* HTTPS yönlendirmesi  
  hepsi burada ayarlanır.

**🧩 1. using Satırları**

| **Satır** | **Görevi** |
| --- | --- |
| using Microsoft.EntityFrameworkCore; | EF Core (Entity Framework Core) kütüphanesini kullanmamızı sağlar. Veritabanı bağlantısı için gerekli. |
| using RestApi.Utils; | Proje içinde yazdığın ApplicationDbContext sınıfı büyük olasılıkla bu klasörün (Utils) içinde. Onu dahil eder. |

**🏗️ 2. var builder = WebApplication.CreateBuilder(args);**

Bu satır uygulamayı başlatır ve **bağımlılık konteynerini (Dependency Injection Container)** oluşturur. Yani:

* builder nesnesi üzerinden servisleri ekleriz (AddControllers, AddDbContext vs),
* Yapılandırmaları (appsettings.json) okur,
* Web uygulaması pipeline’ını hazırlar.

Kısaca: **uygulama yapılandırmasını başlatır.**

**🌐 3. Swagger (OpenAPI) Ekleme**

**builder.Services.AddOpenApi();** Bu satır, **Swagger/OpenAPI** dokümantasyonunu aktif hale getirir.

| **Terim** | **Açıklama** |
| --- | --- |
| **Swagger** | API’leri test etmek ve dökümantasyon oluşturmak için kullanılan bir arayüzdür. |
| **OpenAPI** | REST API’lerin standart tanımlama formatıdır. |

➡️ Bu özellik sayesinde tarayıcıda https://localhost:5001/swagger gibi bir adrese gidip API’leri test edebilirsin.

**🧰 4. Veritabanı Bağlantısı (DbContext)**

**builder.Services.AddDbContext<ApplicationDbContext>(option =>**

**{**

**var path = builder.Configuration.GetConnectionString("DefaultConnection");**

**option.UseSqlite(path);**

**});** Bu kısım çok önemli!

Burada **Entity Framework Core veritabanı bağlantısı** yapılandırılıyor.

| **Kod** | **Açıklama** |
| --- | --- |
| AddDbContext<ApplicationDbContext> | EF Core’a hangi **DbContext sınıfını** kullanacağını söylüyor. |
| builder.Configuration.GetConnectionString("DefaultConnection") | appsettings.json dosyasındaki DefaultConnection bağlantı cümlesini alır. |
| option.UseSqlite(path) | Kullanılacak **veritabanı motorunu (provider)** belirtir. Burada SQLite seçilmiş. |

📌 Yani bu kod, uygulamanın SQLite veritabanına bağlanmasını sağlar. Entity Framework, ApplicationDbContext içinde tanımlı modelleri alır ve tabloya dönüştürür.

**🔹 Örnek: appsettings.json içinde ne vardır?**

Genelde şu şekilde olur:

**"AllowedHosts": "\*",**

**"ConnectionStrings": {**

**"DefaultConnetcion": "Data Source=restApi.db"}**

**}**

**🧱 5. Controller Servislerini Ekleme**

**builder.Services.AddControllers();**

Bu satır, **MVC Controller** sınıflarını uygulamaya ekler.  
Yani senin oluşturduğun UserController, ProductController gibi API controller’ları bu sayede tanınır.

💡 Controllers, dış dünyadan gelen HTTP isteklerini (GET, POST, PUT, DELETE) işler.

**⚙️ 6. Uygulamanın Oluşturulması**

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

Bu satır, az önce oluşturduğumuz tüm yapılandırmaları (builder içinde tanımlanan servisleri) **uygulama nesnesine** dönüştürür.

Artık app nesnesi üzerinden:

* Middleware’leri ekleyebiliriz,
* Endpoint’leri tanımlayabiliriz.

**📘 7. Swagger Haritalama (MapOpenApi)**

**app.MapOpenApi();**

Bu satır, Swagger arayüzünü aktif hale getirir. Uygulama çalıştığında Swagger sayfasına gidip API’lerini görsel olarak test edebilirsin.

**🔒 8. HTTPS Yönlendirmesi**

**app.UseHttpsRedirection();**

Bu satır, tüm HTTP isteklerini otomatik olarak **HTTPS’e yönlendirir.**

➡️ Güvenlik sağlar.  
➡️ Özellikle kullanıcı verisi (örneğin login, register) içeren API’lerde zorunludur.

**🧭 9. Controller Rotalarını Eşleme**

**app.MapControllers();**

Bu satır, **controller’larda tanımladığın tüm route’ları** (örneğin [Route("api/[controller]")]) aktif hale getirir. Yani:

* GET /api/user
* POST /api/user  
  gibi yollar artık çalışır hale gelir.

**▶️ 10. Uygulamayı Çalıştırma**

**app.Run();**

Bu komut, uygulamayı başlatır. Artık web sunucusu dinlemeye başlar (örneğin https://localhost:5001).

**🔁 Özet Akış**

Aşağıdaki tablo, Program.cs dosyasının **işleyiş sırasını** özetler:

| **Aşama** | **Ne Yapıyor** | **Örnek Kod** |
| --- | --- | --- |
| 1️⃣ | Builder oluşturulur | WebApplication.CreateBuilder(args) |
| 2️⃣ | Swagger eklenir | builder.Services.AddOpenApi() |
| 3️⃣ | Veritabanı bağlantısı kurulur | AddDbContext<ApplicationDbContext> |
| 4️⃣ | Controller servisi eklenir | AddControllers() |
| 5️⃣ | Uygulama oluşturulur | builder.Build() |
| 6️⃣ | Swagger aktif edilir | app.MapOpenApi() |
| 7️⃣ | HTTPS yönlendirmesi yapılır | app.UseHttpsRedirection() |
| 8️⃣ | Controller route’ları tanımlanır | app.MapControllers() |
| 9️⃣ | Uygulama çalışır | app.Run() |

**💡 Gerçek Hayatta Bu Dosya Ne İşe Yarar?**

Bu dosya:

* Uygulamanın **başlatıldığı**,
* Servislerin **kayıt edildiği**,
* Middleware zincirinin **oluşturulduğu**,
* Ve **HTTP sunucusunun ayağa kaldırıldığı** yerdir.

Örneğin:

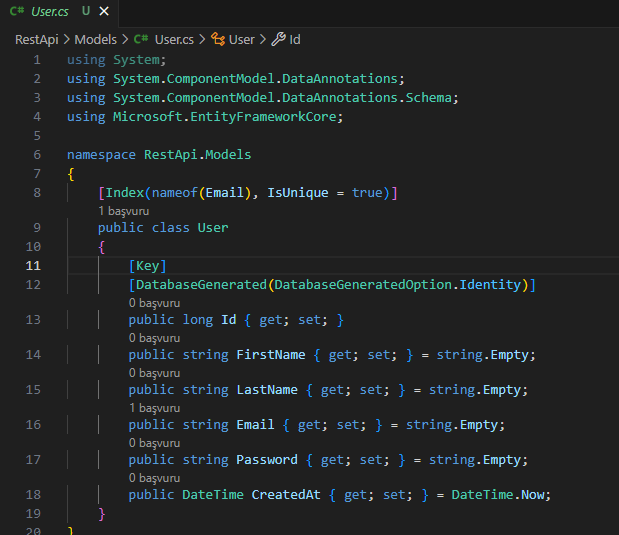
* UserController üzerinden kullanıcı kayıt işlemi yapılırken,
* ApplicationDbContext aracılığıyla User modeli veritabanına kaydedilir,
* Program.cs bu yapıların birbirine bağlanmasını sağlar.

Yani, **uygulamanın kalbi gibidir** ❤️

User.cs dosyası, **Entity Framework Core (EF Core)** kullanan bir **model sınıfıdır**. Bu sınıf, veritabanındaki **Users** tablosunun birebir temsilini (mapping) yapar.

**📘 User.cs Dosyasının Amacı**

Bu dosya, sistemdeki kullanıcıların (user) bilgilerini tutmak için oluşturulmuş bir **Entity (varlık)** sınıfıdır. EF Core bu sınıftaki property’leri kullanarak **veritabanında bir tablo oluşturur.**



**🔹 Üst Kısımdaki using İfadeleri:** Bunlar gerekli kütüphane referanslarıdır:

| **Satır** | **Açıklama** |
| --- | --- |
| using System; | Temel .NET sınıflarını kullanmamızı sağlar (örneğin DateTime). |
| using System.ComponentModel.DataAnnotations; | [Key], [Required], [StringLength] gibi veri doğrulama (validation) ve kimliklendirme özniteliklerini sağlar. |
| using System.ComponentModel.DataAnnotations.Schema; | [DatabaseGenerated], [Column], [ForeignKey] gibi veritabanı ile ilgili öznitelikleri içerir. |
| using Microsoft.EntityFrameworkCore; | EF Core’un özelliklerini kullanmak için gereklidir (örneğin [Index] gibi). |

**🧱 Namespace ve Sınıf Tanımı**

**namespace RestApi.Models**

**{ [Index(nameof(Email), IsUnique = true)]**

**public class User**

| **Kısım** | **Açıklama** |
| --- | --- |
| namespace RestApi.Models | Bu sınıfın proje içindeki konumunu belirtir. Genellikle “Models” klasöründeki tüm entity sınıfları burada yer alır. |
| [Index(nameof(Email), IsUnique = true)] | Veritabanında **Email sütununa unique (benzersiz)** bir index ekler. Aynı e-posta ile iki kullanıcı kayıt olamaz. |
| public class User | “User” adında bir sınıf tanımlar. EF Core bunu bir tabloya dönüştürür (tablo adı otomatik olarak Users olur). |

**📋 Property Açıklamaları**

**🆔 1. Kimlik Alanı (Id)**

**[Key]**

**[DatabaseGenerated(DatabaseGeneratedOption.Identity)]**

**public long Id { get; set; }**

| **Özellik** | **Açıklama** |
| --- | --- |
| [Key] | Bu alanın tablodaki **primary key (birincil anahtar)** olduğunu belirtir. |
| [DatabaseGenerated(DatabaseGeneratedOption.Identity)] | Veritabanı bu değeri **otomatik olarak artırır (auto-increment)**. |
| public long Id | Kimlik numarası (örneğin 1, 2, 3, ...). **long** türü sayesinde çok fazla kullanıcı desteklenebilir. |

**🧍‍♂️ 2. Ad Soyad Alanları**

**public string FirstName { get; set; } = string.Empty;**

**public string LastName { get; set; } = string.Empty;**

| **Alan** | **Açıklama** |
| --- | --- |
| FirstName | Kullanıcının adı. Boş değer ataması engellemek için varsayılan olarak string.Empty verilmiş. |
| LastName | Kullanıcının soyadı. Aynı şekilde boş değer önlenmiş. |

💡 Bu iki alan UI tarafında (örneğin kayıt formu) genellikle zorunlu yapılır.

**✉️ 3. E-posta Alanı**

**public string Email { get; set; } = string.Empty;**

* Kullanıcının sisteme giriş yaparken kullanacağı **benzersiz (unique)** e-posta adresidir.
* [Index(nameof(Email), IsUnique = true)] sayesinde aynı e-posta adresiyle ikinci kez kayıt yapılamaz.
* Uygulama tarafında genellikle [Required] ve [EmailAddress] attribute’ları da eklenir.

**🔐 4. Şifre Alanı**

**public string Password { get; set; } = string.Empty;**

* Kullanıcının parolasını tutar.
* Gerçek projelerde **şifre asla düz metin (plain text)** olarak saklanmaz!
* Bunun yerine genellikle **hash’lenmiş** olarak saklanır (örneğin bcrypt veya SHA256 kullanılarak).

Örnek: Password = BCrypt.HashPassword(plainPassword);

**⏰ 5. Kayıt Zamanı**

public DateTime CreatedAt { get; set; } = DateTime.Now;

* Kullanıcının sisteme kaydolduğu zamanı otomatik olarak alır.
* DateTime.Now ifadesi, kullanıcı oluşturulduğu anda geçerli tarihi ve saati yazar.
* Bu sütun veritabanında datetime veya timestamp türünde tutulur.

Böylece, her kullanıcının **kayıt tarihi** sistemde otomatik olarak saklanır.

**🧠 Özetle**

| **Alan** | **Görevi** | **Ek Bilgi** |
| --- | --- | --- |
| Id | Birincil anahtar | Otomatik artar |
| FirstName | Kullanıcının adı | Boş olamaz |
| LastName | Kullanıcının soyadı | Boş olamaz |
| Email | Giriş için e-posta | Unique index ile korunur |
| Password | Kullanıcının şifresi | Hash’lenerek saklanmalı |
| CreatedAt | Kayıt tarihi | Otomatik olarak atanır |

**💡 Ek Bilgiler (Not olarak ekle)**

* EF Core bu sınıftan otomatik olarak bir tablo oluşturur: **Users**
* Tablo sütunları property isimleriyle eşleşir.
* [Key] -> Primary Key
* [Index] -> Unique veya sıralama kolaylığı sağlar
* [DatabaseGenerated] -> Identity (auto increment) özelliklerini belirler

**🧩 Gerçek Hayatta Kullanımı**

Bu User modeli genellikle şu amaçlarla kullanılır:

* **Kayıt (Register)** işleminde formdan gelen kullanıcıyı veritabanına eklemek.
* **Giriş (Login)** işleminde e-posta ve şifre doğrulaması yapmak.
* **Profil güncelleme, silme** gibi işlemleri yönetmek.
* **JWT (JSON Web Token)** ile kimlik doğrulama sistemlerinde kullanıcıyı temsil etmek.

**📘 1️⃣ Migration (Göç) Nedir?**

Migration, Entity Framework Core’da **veri tabanı şemasını (tablolar, kolonlar, ilişkiler)** kod üzerinden yönetmemizi sağlayan sistemdir. Yani sen tabloyu **SQL komutlarıyla** oluşturmazsın; sadece bir **C# model sınıfı** (User.cs gibi) yazarsın, EF Core bunu analiz eder ve veri tabanında karşılık gelen yapıyı (örneğin Users tablosu) otomatik oluşturur.

**⚙️ 2️⃣ Migration Neden Kullanılır?** Eskiden tabloyu oluşturmak için elle SQL yazardık:

**CREATE TABLE Users (**

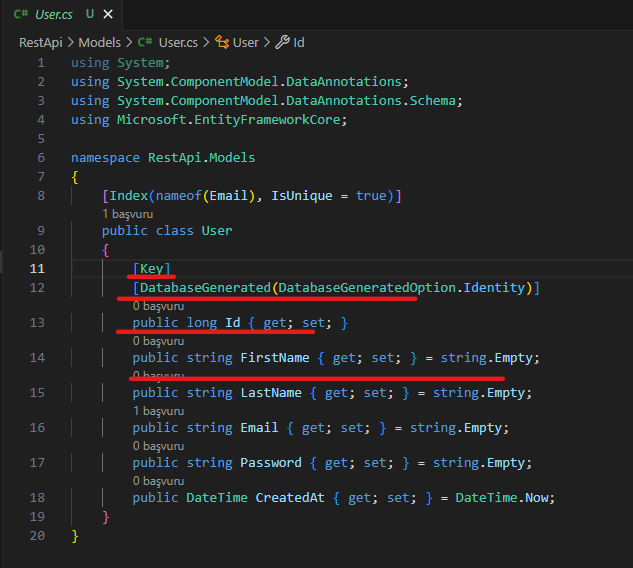
**Id INTEGER PRIMARY KEY,**

**FirstName TEXT,**

**LastName TEXT**

**);**

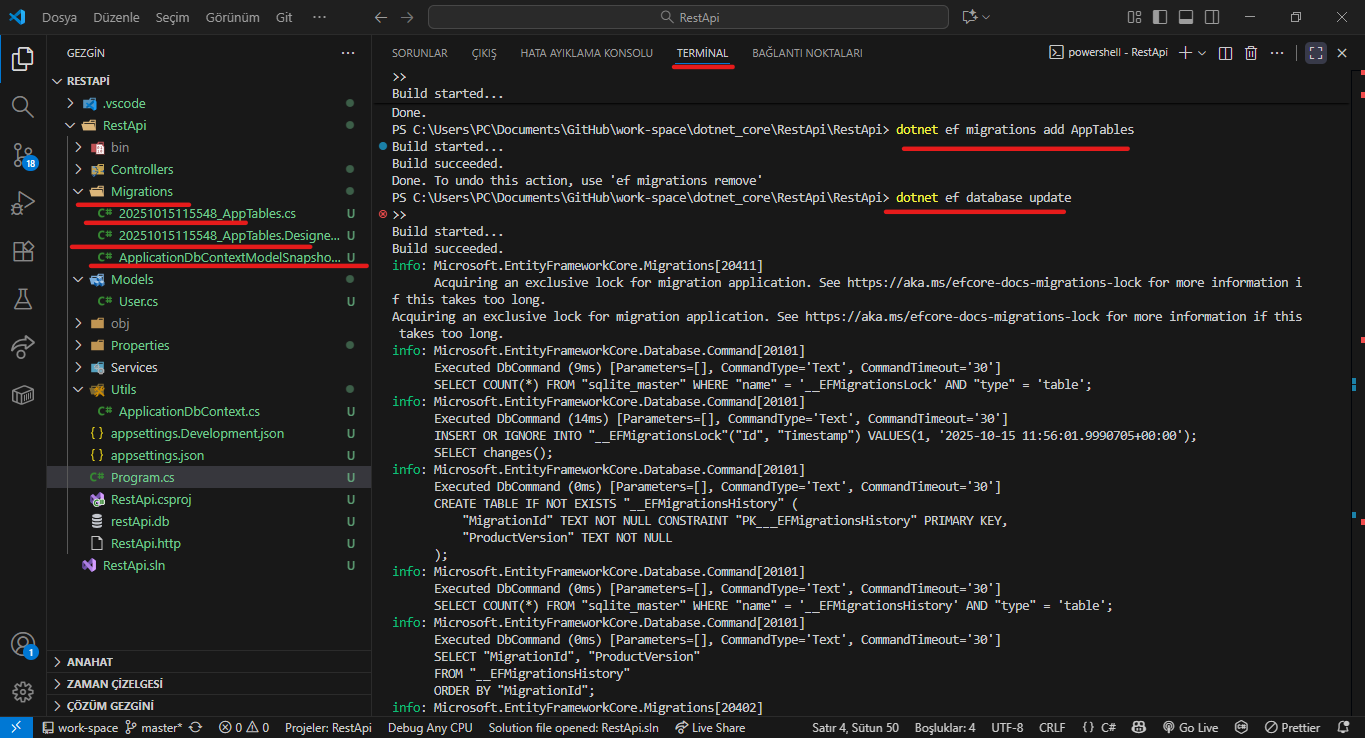
Ama EF Core sayesinde artık sadece şunu yazıyorsun:



Ve ardından terminalde:

**dotnet ef migrations add CreateUserTable**

**dotnet ef database update** bu iki komutla EF Core arkada bu SQL’i **otomatik** üretir ve veritabanına uygular. Yani **migration = SQL’in C# versiyonu + sürüm kontrol sistemi** diyebilirsin. 🔁



**🧩 3️⃣ Migration’ın Veritabanıyla İlişkisi**

Migration sistemi **3 katmanlı** çalışır:

| **Katman** | **Açıklama** |
| --- | --- |
| 🧠 **Model Katmanı** | Kod tarafındaki C# sınıfların (User.cs, Book.cs, vs.). |
| 🗃️ **Migration Katmanı** | EF Core’un bu modelleri veritabanına çevirdiği geçiş (göç) dosyaları. |
| 💾 **Database Katmanı** | Migration’lar sonunda oluşan gerçek SQLite veritabanı (restApi.db). |

Bu üçü daima **senkron** olmalıdır. Birinde değişiklik olursa (örneğin modele yeni property eklersen), EF Core diğerlerini güncellemeni ister.

**🧠 4️⃣ Migration’ın İşleyiş Adımları**

1️⃣ **Model oluşturulur**  
(User.cs, Book.cs gibi sınıflar tanımlanır)

2️⃣ **Migration eklenir**

**dotnet ef migrations add CreateUserTable**

→ EF Core modeldeki tablo tanımlarını okur  
→ SQL komutları üreten bir migration dosyası oluşturur (örneğin CreateTable kodları içerir)

3️⃣ **Database güncellenir**

**dotnet ef database update**

→ EF Core migration dosyasını çalıştırır  
→ SQLite veritabanına tabloyu gerçekten ekler.

**🧮 5️⃣ Migration Dosyaları Ne İçerir?**

**Örnek:**

**protected override void Up(MigrationBuilder migrationBuilder)**

**{**

**migrationBuilder.CreateTable(**

**name: "Users",**

**columns: table => new**

**{**

**Id = table.Column<long>(nullable: false)**

**.Annotation("Sqlite:Autoincrement", true),**

**FirstName = table.Column<string>(nullable: false),**

**LastName = table.Column<string>(nullable: false)**

**},**

**constraints: table =>**

**{**

**table.PrimaryKey("PK\_Users", x => x.Id);**

**});**

**}**

* Up() metodu → **veritabanında tabloyu oluşturur**
* Down() metodu → **geri alınırsa tabloyu siler** yani migration’lar, **“ileri gitme” (Up) ve “geri dönme” (Down)** mekanizmasıdır.

**🧱 6️⃣ Migration’ların Sürüm Mantığı:** Her migration dosyasının adı zaman damgasıyla başlar:

**20251015120000\_CreateUserTable.cs**

**20251015123000\_AddEmailColumn.cs**

EF Core bu sırayı takip eder. İstersen geçmiş migration’a geri dönebilirsin:

**dotnet ef database update 20251015120000\_CreateUserTable** → yeni eklenen sütunları geri alır.

**🔍 7️⃣ Snapshot Dosyası Nedir?**

**ApplicationDbContextModelSnapshot.cs**

Bu dosya EF Core’un “şu anki modelin fotoğrafı” gibidir.  
Yeni migration eklerken EF Core, modeli bu snapshot ile karşılaştırır ve farkı bulur.

**🧰 8️⃣ Migration Günlük Hayatta Ne İşe Yarar?**

* Geliştiriciler ekip halinde çalışırken her biri tabloya alan ekleyebilir.  
  Migration sistemi sayesinde herkesin değişikliği kayda geçer.
* Geliştirme → test → üretim ortamına taşırken veritabanı şeması otomatik güncellenir.
* SQL yazmadan, güvenli ve sürümlü şekilde veri modeli yönetilir.

**🧭 9️⃣ En Temel Komutlar (özet)**

| **Komut** | **Görev** |
| --- | --- |
| dotnet ef migrations add <Ad> | Yeni migration oluşturur |
| dotnet ef database update | Migration’ları veritabanına uygular |
| dotnet ef migrations remove | Son migration’ı kaldırır |
| dotnet ef migrations list | Mevcut migration’ları listeler |
| dotnet ef database update <MigrationAdı> | Belirli migration’a geri döner |

**💡 Özet**

Migration, **C# modelleri ile veri tabanı arasında köprü** kurar. EF Core, kodda tanımlanan sınıfları migration’lar aracılığıyla SQL’e çevirir ve senkron tutar. Böylece geliştirici SQL yazmadan veri tabanı yönetimi yapabilir.