# MVC projesi terminal komutları

* dotnet –version
* docker ps çalışan konteynerleri gösterir.
* cd ~/Desktop hangi dizinde olmak istediğine göre değiştirilebilir.
* mkdir MVCTicariOtomasyon , cd MVCTicariOtomasyon

bu komutlarla klasör oluşur ve içine erişim sağlanır.

* MVCTicariOtomasyon klasörü içinde terminalden dotnet new mvc -n MVCTicariOtomasyon komutu çalıştır. Bu komut MVCTicariOtomasyon adında bir ASP.NET MVC projesi oluşturur. İçinde otomatik olarak Controllers, Views, Models klasörlerini getirir.
* ls alt klasörleri gösterir.
* Ctrl + C ile işlem durur.
* cd Models/sınıflar alt klasörlere gitmek için gereken komut yazımı.
* touch Kategori.cs yeni bir C# dosyası oluşturur.
* ~ kullanıcınını ana dizinidir. Yani /Users/ahsennuraldas

Örneğin cd ~ 🡪 Ana klasöre git 🡪 /Users/ahsennuraldas dizinine döner.

* Entity Framework’ü (EF Core) paket olarak projeye eklemek için şu komutlar yazılır:  
  dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

* EntityFrameworkCore ‚ EF'in ana kütüphanesi; SqlServer ‚ SQL Server ile iletişim sağlar(biz Docker'daki SQL'e bağlanacağız); Tools, Migration (veritabanı oluşturma/güncelleme) komutlarını kullanmayı sağlar. Bunlar yüklenince "References" bölümüne denk olan bağımlılıklar artık MVCTicariOtomasyonWeb.csproj içine otomatik eklenmiş olur.
* Context sınıfı Entity Framework’ün veritabanı ile konuştuğu beyin gibidir. Yani EF tablolara buradan ulaşır, DbSet tanımları buraya yazılır.

|  |  |
| --- | --- |
| Eski yapı (MVC 5) | Yeni yapı (.NET Core) |
| `web.config` içinde bağlantı | `appsettings.json` içinde bağlantı |
| EF6 (`System.Data.Entity`) | EF Core |
| `DbContext` doğrudan connection string bulur. | Connection string Startup/Program.cs üzerinden verilir |

* Migration = **veritabanı ile C# modellerin (sınıfların) arasındaki farkı senkronize eden araçtır.** Yani EF Core şunu yapar:   
  “Senin kodunda hangi tablolar (DbSet) ve alanlar (property) varsa, onları veritabanında da oluşturayım.”   
  Bunu **adım adım izlenebilir bir sistemle** yapar. İşte bu “adım” dediğimiz şey **migration** dosyasıdır.

Neden Migration Dosyası Oluşturuyoruz? Çünkü:

Kod tarafında modelini (örneğin Kategori, Urun, Personel) değiştirebilirsin.  
EF Core da bu değişiklikleri takip eder. Ama veritabanı, sen söylemedikçe kendini güncellemez. İşte EF’ye “bu değişiklikleri veritabanına yansıt” demek için **migration dosyası oluştururuz. Migration dosyaları EF tarafından otomatik oluşturulur.**

* dotnet build komutu projeni derler ve bize tam hata mesajlarını gösterir.
* dotnet ef database update 🡪 Docker’daki SQL Server’a bağlanır,

migration dosyandaki CreateTable komutlarını çalıştırır, gerçek veritabanını (MVCTicariOtomasyon) oluşturur, içine \_\_EFMigrationsHistory tablosunu ekler (migration geçmişini tutar).

* dotnet ef database update 0 komutu veritabanını migration öncesine (başlangıç hâline) döndürür. Tüm tabloları siler, veritabanı sıfır hale gelir (schema sıfırlanır ama class’lar durur).
* dotnet ef migrations remove komutu VeritabaniKurulumu migration dosyasını projeden kaldırır. Eğer birden fazla migration varsa, bunu birkaç kez üst üste çalıştırmak gerekir. (ta ki “No migrations found” mesajı çıkana kadar)
* dotnet ef migrations add InitialCreate

dotnet build

komutları yeni migration oluşturur. dotnet ef database update komutu ile de oluşan migration veritabanına uygulanır.

* rm -rf Migrations Bu komut Migrations klasörünü komple siler.
* Veritabanına dokunmaz — tablolar, veriler aynı kalır.