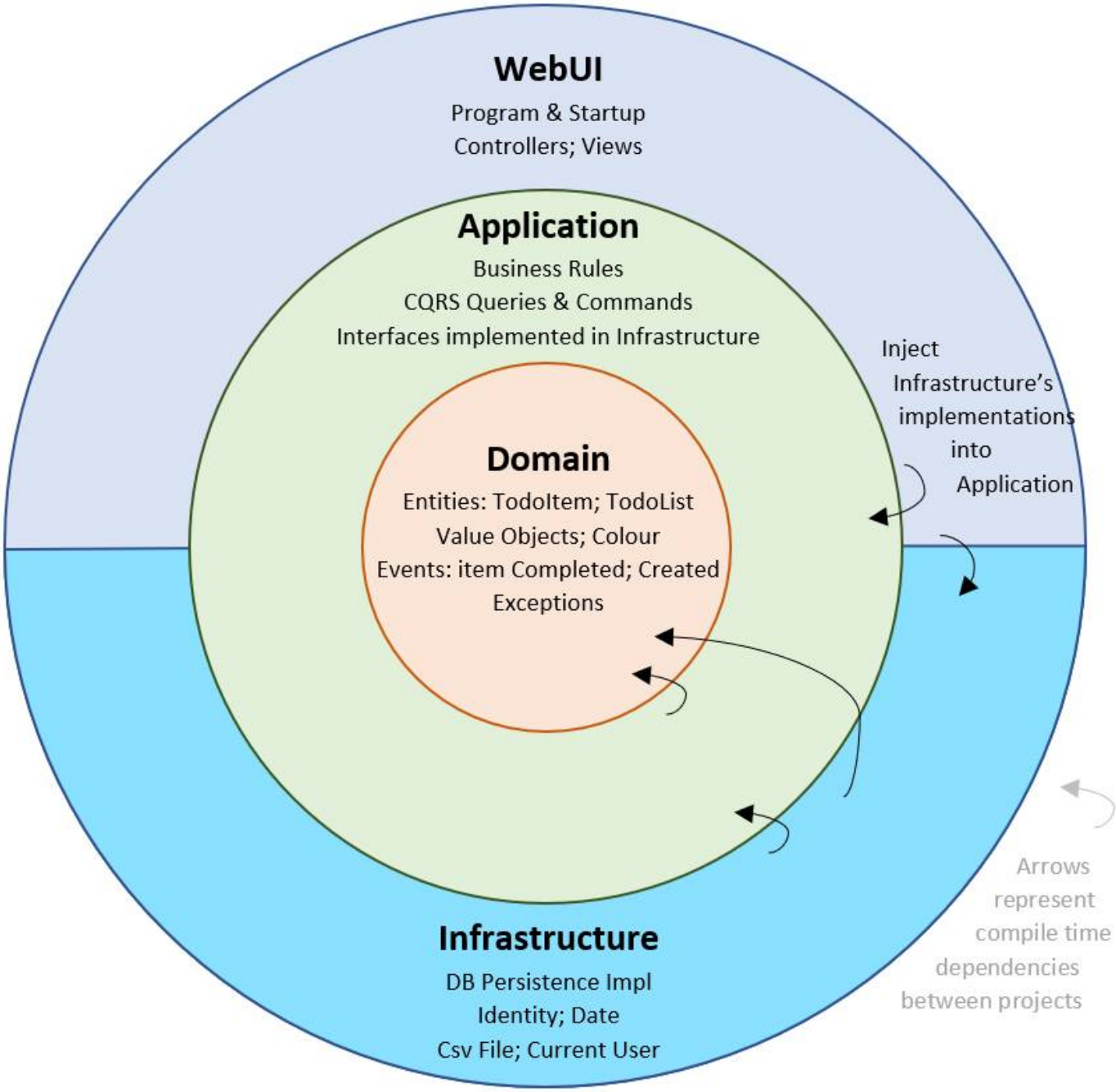
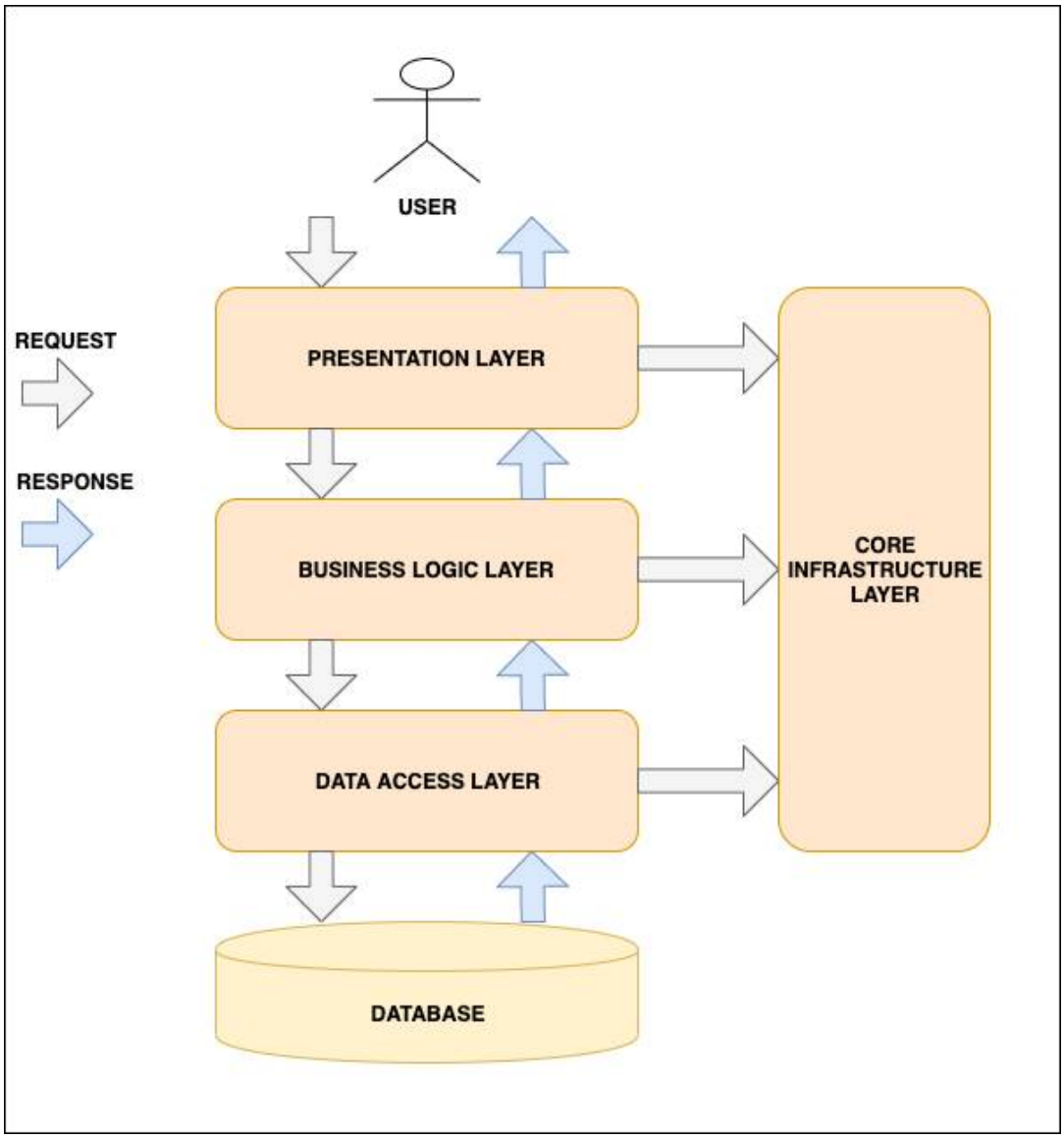
Dotnet Core katmanları

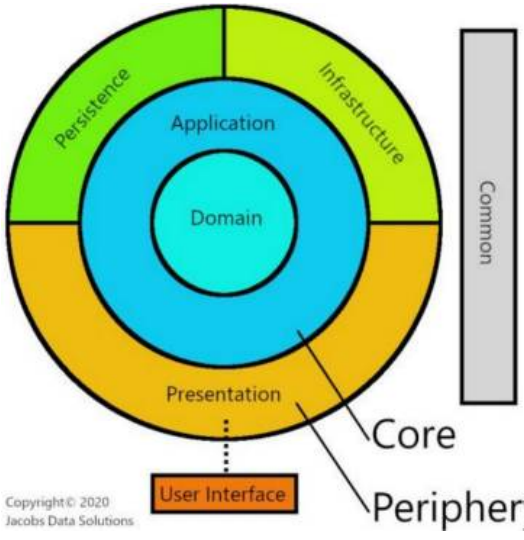
dotner core ile restapi projesi yazacağım katmanlara ayrılmış olmasını istiyorum hangi katmanlara ihtiyacım var ve altında hangi davranışları bulundurmam gerekiyor

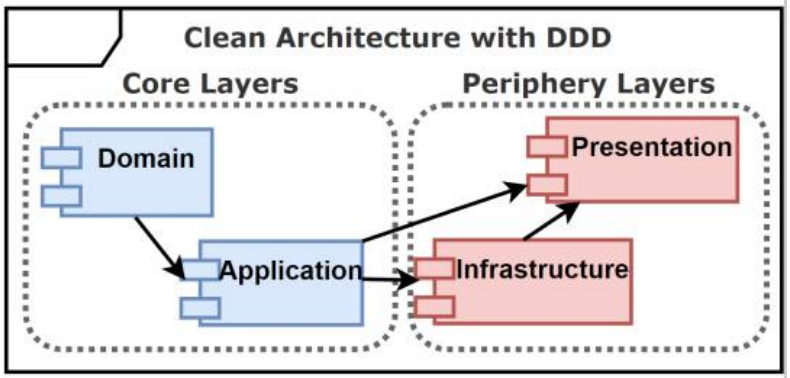
Aşağıda **.NET Core ile kurumsal düzeyde, katmanlı bir REST API projesi** geliştirmek için kullanabileceğin **ideal mimari katmanları** ve **her katmanın içinde bulunması gereken davranışları** eksiksiz şekilde listeledim. Bu yapı, gerçek kurumsal projelerde kullanılan temiz mimari (Clean Architecture) + DDD + SOLID ilkelerine uygundur.

✅ .NET Core REST API İçin Önerilen Katmanlar ve Davranışlar


WebUI
Program & Startup
Controllers; Views
Application
Business Rules
CQRS Queries & Commands
Interfaces implemented in Infrastructure
Domain
Entities: TodoItem; TodoList
Value Objects; Colour
Events: item Completed; Created
Exceptions
Inject
Infrastructure's
implementations
into
Application
Infrastructure
DB Persistence Impl
Identity; Date
Csv File; Current User
Arrows
represent
compile time
dependencies
between projects







🎯 1. API Layer (Presentation Layer)

Projenin dış dünya ile konuşan yüzüdür.

📌 İçermesi Gerekenler

- **Controller** sınıfları
- **Action** metotları
- HTTP Request/Response yönetimi
- Swagger/OpenAPI config
- Model binding + validation (FluentValidation tercih edilir)
- API versiyonlaması
- Authentication / Authorization (JWT, Policy vb.)
- Global exception handling (Middleware)

📌 İçermemesi Gerekenler

- **Business logic**
- **Database işlemleri**
- **Entity** kullanımı (DTO kullanılmalı)

🎯 2. Application Layer

Tüm iş kuralları ve uygulama süreçlerinin olduğu katmandır.
Genelde CQRS + MediatR kullanılır.

📌 İçermesi Gerekenler

- **Command & Query handler'lar**
- **DTO / Request / Response modelleri**
- **Business Rules (Domain'i kirletmeyen kurallar)**
- **Service Interface'leri**
- **Mapping profilleri (AutoMapper)**
- **Validation (FluentValidation)**

📌 Yapmaması Gerekenler

- Database erişimi
- Infrastructure bağımlılığı (ör. EF DbContext kullanamaz)

🎯 3. Domain Layer

Mimarinin kalbi. Saf C# kodu.

📌 İçermesi Gerekenler

- Entity sınıfları

- Value Object'ler
- Domain Events
- Repository Interface'leri
- Aggregate Root'lar
- Temel domain kuralları (örn. fiyat < 0 olamaz)

📌 Yapmaması Gerekenler

- Veritabanı kodları
- API/DTO bağımlılığı
- Framework bağımlılığı (EF Core attribute'ları bile minimum olmalı)

🎯 4. Infrastructure Layer

Teknolojiye bağlı her şey buradadır.

📌 İçermesi Gerekenler

- EF Core DbContext
- Repository implementasyonları
- Migrations
- Email servisleri (SMTP)
- Cache servisleri (Redis)
- File storage
- Logging (Serilog)
- Queue'lar (RabbitMQ, Kafka)
- Payment provider bağlantıları
- Dış API entegrasyonları

📌 Yapmaması Gerekenler

- İş kuralları
- Domain manipülasyonu

🎯 5. Persistence Layer (opsiyonel ama güçlü mimarilerde ayrılır)

Infrastructure'ın özel bir alt katmanı olarak düşünebilirsin.

📌 İçermesi Gerekenler

- EF Core configuration dosyaları (`IEntityTypeConfiguration`)

- DbContext
- Repository implementasyonları
- Unit of Work

🎯 6. Common/Shared Layer (opsiyonel)

Ortak kullanılan componentler buraya alınır.

📌 İçermesi Gerekenler

- Custom exceptionlar
- Result wrapper (SuccessResult, ErrorResult vb.)
- BaseEntity
- Extensions
- JWT helper

📌 Genel Katmanlaşma Örneği (Final Yapı)

```
scss

src
 ├── API
 │    └── Controllers
 │    └── Filters
 │    └── Middlewares
 │    └── Models (DTO)
 │
 ├── Application
 │    └── Commands
 │    └── Queries
 │    └── Interfaces
 │    └── Validators
 │    └── Behaviors (Logging, Caching)
 │
 ├── Domain
 │    └── Entities
 │    └── ValueObjects
 │    └── Events
 │    └── Interfaces (IRepository gibi)
 │
 ├── Infrastructure
 │    └── Persistence (DbContext, Repository)
 │    └── Services (Mail, Cache)
 │    └── Logging
 │
 └── Shared (opsiyonel)
      └── Exceptions
      └── Results
```

📘 İstersen Sana Bu Yapıda Bir Sample Proje Oluşturayım

Tam böyle bir mimaride hazır:

- Layered + Clean Architecture
- CQRS + MediatR
- EF Core + SQL
- JWT Authentication
- Logging + Exception middleware

İstersen klasör klasör çıkartayım veya **zip şeklinde sıfırdan bir proje** oluşturup paylaşayım.

İstersen hangisini istediğini söyle 👍

aşağıdaki senaryoyu, adım adım hangi katmanda olması gerektiğini detaylı açıkla

Kullanıcı login;
Kullanıcı login için user json nesnesi geldi
dto ya dönüştü, model mapper kullan
email doğrulama yaptık
başarısız ise global bir hata mesajı sunduk
başarılı ise veri tabanına yazdık
global response ile sonuç dönderdik

Harika — aşağıdaki açıklama, verdiğin **login senaryosunu adım adım** alır ve **her bir davranışın hangi katmanda** olması gerektiğini, neden öyle olması gerektiğini ve küçük, işe yarar örnek kod/snippet'larla gösterir. Türkçe, net ve uygulamaya hazır bir yol haritası:

1. Akışın kısa özeti (adımlar)

1. Controller (API) gelen JSON'u alır.
2. DTO validasyonu yapılır (Application veya API layer).
3. DTO → Domain model/Command dönüşümü (AutoMapper) — Application.
4. Email doğrulama (business rule) — Domain veya Application (hangisi domain kuralıysa).
5. Doğrulama başarısızsa hata fırlatılır veya Result döndürülür — Application / Domain, Global Exception Middleware API'de yakalar.
6. Başarılıysa Repository üzerinden veritabanına yazılır — Infrastructure (Persistence).
7. Uygulama bir token üretmesi gerekiyorsa bunu Infrastructure / Application yapar (JWT servis).
8. Sonuç global response wrapper ile API'ye döndürülür — API layer (veya Application dönüş formatı).

---

2. Detaylı katmanlara göre görev dağılımı

API Layer (Presentation) — `AuthController`

- **Gelen request**: raw JSON body → **`LoginRequestDto`**.
- **Model binding & hızlı doğrulama**: (isteğe bağlı) temel JSON/format doğrulaması. Ancak iş kuralları Application'da olmalı.
- **Kullanacağı şeyler**: **`IMediator`/`IAuthService`** çağrısı, global exception middleware ile hataları dışa vermeme.
- **Dönüş**: Global **`ApiResponse<T>`** wrapper ile HTTP status ve payload.

Örnek:

```csharp
[ApiController]
[Route("api/auth")]
public class AuthController : ControllerBase
{
    private readonly IMediator _mediator;
    public AuthController(IMediator mediator) => _mediator = mediator;

    [HttpPost("login")]
    public async Task<IActionResult> Login([FromBody] LoginRequestDto dto)
    {
        var result = await _mediator.Send(new LoginCommand(dto));
        return StatusCode(result.StatusCode, ApiResponse.From(result));
    }
}
```

Application Layer — `LoginCommand`, Handler, Validation, Mapping

- **DTO → Command**: **`LoginRequestDto`** map edilir (**`AutoMapper`**) → **`LoginCommand`** veya **`LoginRequest`** model.
- **Validation**: FluentValidation burada; örneğin email format, password boş mu gibi temel kontroller. (Daha kapsamlı iş kuralları Domain'de).
- **Handler**: Tüm süreçin orkestrasyonu burada gerçekleşir:
  - **`IUserRepository.GetByEmail(email)`** çağrılır.
  - Eğer user yoksa veya şifre doğrulanamazsa **`Result.Failure(...)`** döndür.
  - Email doğrulama bir domain kuralıysa Domain servis çağrılır veya Application içerisinde **`IIdentityService.VerifyEmail(...)`** kullanılır.
  - Başarılıysa **`IUserRepository.AddOrUpdate(...)`** veya **`SaveChangesAsync()`** çağrılır (Unit of Work).
  - Token oluşturma için **`ITokenService`** çağrılır (genelde Infrastructure tarafından uygulanır, interface Application içinde).
  - **`LoginResultDto`** oluşturulur ve döndürülür.

Örnek Handler (kısaltılmış):

```csharp
public class LoginCommandHandler : IRequest<Result<LoginResultDto>>
{
    private readonly IUserRepository _repo;
    private readonly ITokenService _tokenService;
    private readonly IMapper _mapper;

    public async Task<Result<LoginResultDto>> Handle(LoginCommand cmd, CancellationToken ct)
    {
        // 1. Get user
        var user = await _repo.GetByEmailAsync(cmd.Email);
        if (user == null) return Result.Failure<LoginResultDto>("Kullanıcı bulunamadı.");

        // 2. Verify password (domain rule)
        if (!user.VerifyPassword(cmd.Password))
            return Result.Failure<LoginResultDto>("Email veya şifre hatalı.");

        // 3. Email doğrulaması domain kuralıysa kontrol
        if (!user.IsEmailConfirmed)
            return Result.Failure<LoginResultDto>("Email doğrulanmamış.");

        // 4. (Opsiyonel) Update last login
        user.SetLastLogin(DateTime.UtcNow);
        await _repo.UnitOfWork.SaveChangesAsync();

        // 5. Token oluştur
        var token = _tokenService.GenerateToken(user);

        var dto = new LoginResultDto { Token = token, Name = user.Name };
        return Result.Success(dto);
    }
}
```

Domain Layer — Entity, Business Rules, Value Objects

- **Entity**: **`User`** sınıfı; **`VerifyPassword`**, **`IsEmailConfirmed`**, **`ConfirmEmail`**, **`SetLastLogin`** gibi metotlar domain üzerinde olmalı.
- **Neden**: Email doğrulama ve şifre kontrolü gerçek business rule'dür; domain model içinde kapsüllenmelidir. Bu sayede farklı uygulamalar aynı davranışı tekrar kullanır.
- **Domain Events**: Kullanıcı giriş yaptıktan sonra **`UserLoggedInEvent`** publish edilebilir.

Örnek User metodu:

```csharp
public class User : AggregateRoot
{
    public string Email { get; private set; }
    private string PasswordHash { get; set; }
    public bool IsEmailConfirmed { get; private set; }

    public bool VerifyPassword(string password) => PasswordHasher.Verify(PasswordHash,
password);
    public void ConfirmEmail() => IsEmailConfirmed = true;
    public void SetLastLogin(DateTime dt) { LastLogin = dt; AddDomainEvent(new
UserLoggedInEvent(Id)); }
}
```

Infrastructure Layer — Persistence & External Services

- **Repository implementasyonu**: `UserRepository : IUserRepository` — EF Core kullanarak user sorguları ve update'leri yapar.
- **TokenService (JWT)**: `ITokenService` interface'ini uygular ve secret, expiry bilgisi ile token üretir.
- **UnitOfWork & DbContext**: Transaction yönetimi bu katmanda kalır.
- **Neden**: DB/Framework bağımlılığı domain ve application'dan izole edilir.

Örnek repository metodu:

```
csharp

public async Task<User> GetByEmailAsync(string email) =>
    await _dbContext.Users.SingleOrDefaultAsync(u => u.Email == email);
```

Shared / Common

- **Result / Response wrapper**: `Result<T>` veya `ApiResponse<T>` uygulaması. Başarı / hata / validation bilgileri standardized olur.
- **Global error codes** ve mesaj şablonları burada tutulur.

Örnek:

```
csharp

public class ApiResponse<T>
{
    public bool Success { get; set; }
    public string Message { get; set; }
    public T Data { get; set; }
    public static ApiResponse<T> From(Result<T> r) { ... }
}
```

Global Exception Handling (Middleware) — API Layer

- **Amaç**: Application veya Domain tarafından fırlatılan beklenmeyen hataları yakalamak, kimlik bilgisi sızdırmadan standart bir hata yanıtı döndürmek.
- **Behavior**:
  - `ValidationException` -> 400 + detaylı validation mesajı.
  - `DomainException` -> 400 veya 422 (iş kuralı hatası).
  - `UnauthorizedException` -> 401.
  - Diğer hatalar -> 500 + genel mesaj.
- **Not**: İş kuralları normal akışla Result döndürerek de iletilebilir; exception yalnızca exceptional durumlar için.

Örnek middleware:

```csharp
public async Task Invoke(HttpContext ctx)
{
    try { await _next(ctx); }
    catch (DomainException dex) { await WriteResponse(ctx, 400, dex.Message); }
    catch (Exception ex) { await WriteResponse(ctx, 500, "Beklenmeyen hata oluştu."); }
}
```

---

3. Senaryoya göre adım adım akış (mapping ile)

1. **Client → API**
   - Gönderir: `{ "email":"a@b.com","password":"..." }`
   - API Controller model binding ile `LoginRequestDto` alır.
2. **DTO Validasyonu (Application/Controller)**
   - FluentValidation `LoginRequestDtoValidator` ile email formatı, password boş mu kontrol edilir.
   - Hatalıysa `400` ve validation hataları global pipeline tarafından döner.
3. **DTO → Command/Model (AutoMapper)**
   - `mapper.Map<LoginCommand>(dto)` — Application Layer.
4. **Handler (Application)**
   - `IUserRepository.GetByEmailAsync(email)` çağrılır. (Infrastructure implementasyonu arka planda çalışır)
   - `User` domain objesi döner.
   - `user.VerifyPassword(password)` — **Domain metodu** çağrılır. Eğer false → `Result.Failure("invalid credentials")`.
5. **Email doğrulama**
   - `if (!user.IsEmailConfirmed) return Failure("Email doğrulanmamış")` — Bu kural domain concern ise `user.IsEmailConfirmed` domain property üzerinden okunur; doğrulama action'ı (`ConfirmEmail`) domain içinde yer alır. Eğer email doğrulama tam otomatik değilse `IEmailService` (Infrastructure) ile doğrulama maili tekrar gönderme logic'i Application'da çağrılabilir.
6. **Başarılıysa veritabanına yazma**
   - Örnek: `user.SetLastLogin(DateTime.UtcNow)` — Domain metodu.
   - `await _repo.UnitOfWork.SaveChangesAsync()` — Persistence katmanında transaction ile commit edilir.
7. **Global Response**
   - Handler `Result.Success(new LoginResultDto{ Token = "...", Expires = ... })` döndürür.
   - Controller `ApiResponse` wrapper ile HTTP 200 ve standart JSON döner:

```json
{
  "success": true,
  "message": "Giriş başarılı",
```

```
  "data": { "token": "...", "name": "Ahmet" }
}
```

8. **Başarısız akış örneği**
    - Email doğrulanmamış → Handler **`Result.Failure("Email doğrulanmamış")`** döner.
    - Controller bunu alır ve uygun HTTP status (ör. 400 veya 401) ile **`ApiResponse`** içinde döner.
    - Global Exception Middleware yalnızca beklenmeyen exception'ları (stack trace, 500) ele alır.

---

4. Hangi hatayı nerede fırlatmalı / return edilmeli?

- **Validation hataları (FluentValidation)** → **`400 Bad Request`** — Application doğrulama veya API model validation.
- **Business rule hataları** (e.g. email doğrulanmamış) → **`Result.Failure`** veya **`DomainException`** (tercih edilen: Result pattern; exception performans/akış kontrolü için değil).
- **Sistemsel hatalar (DB down, exception)** → exception fırlatılır ve Global Middleware 500 ile yanıtlar.

---

5. Dosya / namespace önerisi (örnek)

```
swift

MyApp.Api
  /Controllers/AuthController.cs

MyApp.Application
  /Auth/Commands/LoginCommand.cs
  /Auth/Handlers/LoginCommandHandler.cs
  /Dtos/LoginRequestDto.cs
  /Dtos/LoginResultDto.cs
  /Validators/LoginRequestValidator.cs
  /Interfaces/ITokenService.cs

MyApp.Domain
  /Entities/User.cs
  /Events/UserLoggedInEvent.cs

MyApp.Infrastructure
  /Persistence/AppDbContext.cs
  /Repositories/UserRepository.cs
  /Services/JwtTokenService.cs
  /UnitOfWork/UnitOfWork.cs

MyApp.Common
  /ApiResponse.cs
  /Result.cs
```

---

6. Kısa ipuçları & en iyi uygulamalar

- **Password verification**: Hash + salt, **`BCrypt`** veya **`PBKDF2`**. Asla düzmetin şifre saklama.
- **Email doğrulama**: Email doğrulanmamışsa girişe izin verip “email onayı gönderildi” mi göstereceksin yoksa girişi tamamen engelleyecek misin — politika uygulama kararın.

- **Result vs Exception**: İş kurallarında `Result<T>` kullan; exception performans maliyeti ve akış kontrolü için uygun değil.
- **Transaction**: Login sırasında birden fazla değişiklik (ör. audit + lastLogin) yapılıyorsa UnitOfWork ile birlikte DB transaction kullan.
- **Logging**: Başarısız giriş denemelerini rate-limit ile koru ve logla (security).
- **Audit / Security**: Brute-force koruma, account lockout policy, 2FA opsiyonlarını düşün.

İstersen şimdi bu yapıya uygun **tam örnek kod** (API + Application + Domain + Infrastructure) oluşturarak tek bir repo/zip halinde verebilirim — ya da spesifik bir katmanın (`LoginCommandHandler`, `User` entity, `AuthController`, `GlobalExceptionMiddleware`) tam kodunu hemen yazayım. Hangi format istersin?

↪
/Auth/Commands/LoginCommand.cs
bu classs tam olarak nasıl çalışır

Aşağıda **LoginCommand.cs** sınıfının *tam olarak nasıl çalıştığını*, hangi görevi üstlendiğini ve uygulamada nasıl kullanılacağını adım adım, "perde arkası" mantığıyla anlattım. Ayrıca en altta **tam bir örnek dosya** da ekledim.

🎯 LoginCommand Nedir?

Application Layer içinde bulunan bir *komuttur (Command)* ve görevi:

👉 **Controller'dan gelen login isteğini temsil eden bir mesaj / iş emri oluşturmaktır.**

- İş yapmaz
- Veri işlemez
- DB'ye bağlanmaz
- Token üretmez
- Sadece: **login operasyonunun parametrelerini taşır**

Bunu bir "*iş isteği dosyası*" gibi düşün.

🧠 LoginCommand nasıl çalışır? (Flow)

Aşağıdaki örnek akışı takip edelim:

▶️ 1. Client API'ye Login JSON gönderir

```
json

{
  "email": "test@site.com",
  "password": "123456"
}
```

▶️ 2. Controller JSON'u DTO'ya alır

```
csharp

public IActionResult Login([FromBody] LoginRequestDto dto)
{
    return Ok(_mediator.Send(new LoginCommand(dto)));
}
```

Controller **iş mantığı yapmaz** → LoginCommand oluşturup **MediatR**'a yollar.

▶️ 3. LoginCommand Sadece Veriyi Taşır

```
csharp

public class LoginCommand : IRequest<Result<LoginResultDto>>
{
    public string Email { get; set; }
    public string Password { get; set; }

    public LoginCommand(LoginRequestDto dto)
    {
        Email = dto.Email;
        Password = dto.Password;
    }
}
```

- **`IRequest<T>`** → MediatR'a "beni handle edecek bir handler var" demektir.
- Handler dönüş tipini belirler:
  `Result<LoginResultDto>`
- DTO → Command dönüşümü API katmanında yapılır.

▶️ 4. MediatR, LoginCommand için Handler'ı bulur

LoginCommand tek başına hiçbir şey yapmaz.

Asıl iş:

📌 **LoginCommandHandler** içinde gerçekleşir.

```
csharp

public class LoginCommandHandler
    : IRequestHandler<LoginCommand, Result<LoginResultDto>>
{
   // constructor injection

   public async Task<Result<LoginResultDto>> Handle(LoginCommand request, CancellationToken ct)
```

```
    {
        // login business logic burada yapılır
    }
}
```

MediatR şu şekilde çalışır:

1. Controller `Send()` çağırır
2. MediatR → LoginCommand türünü inceler
3. Register edilmiş handler'ı bulur
4. LoginCommand → parametre olarak handler'a verilir
5. Handler login'i gerçekleştirir (repo, token üretme vb.)
6. Handler response (Result<LoginResultDto>) döndürür
7. MediatR → Controller'a geri döner
8. Controller → API Response olarak client'a gönderir

🔍 LoginCommand neden gerekli?

Çünkü **Clean Architecture + CQRS** yaklaşımında:

- Controller'ın işi **komut göndermek**
- Command'ın işi **veri taşımak**
- Handler'ın işi **iş mantığını uygulamak**

Bu ayrım sayesinde:

✔️ Controller temiz kalır
✔️ Handler tek sorumluluğa sahip olur (SRP)
✔️ Test etmesi çok daha kolay olur
✔️ Gereksiz bağımlılıklar API'de oluşmaz
✔️ İş akışı MediatR pipeline davranışlarına uygun hâle gelir (logging, caching, validation otomatik yürür)

🔧 FluentValidation ile Doğrulama Nasıl İşler?

Command'ın yanında **LoginCommandValidator.cs** olur:

```
csharp

public class LoginCommandValidator : AbstractValidator<LoginCommand>
{
    public LoginCommandValidator()
    {
        RuleFor(x => x.Email).NotEmpty().EmailAddress();
        RuleFor(x => x.Password).NotEmpty().MinimumLength(6);
    }
}
```

MediatR pipeline şu şekilde çalışır:

1. Controller → Command gönderir
2. MediatR → Validator'ı bulur
3. Validator hata üretirse → Exception atar
4. Global exception middleware → 400 BadRequest döndürür
5. Handler hiç çalışmaz

Böylece "email boşsa DB'ye bile gitmez."

🔥 Şimdi OKUNABİLİR TAM ÖRNEK: LoginCommand.cs

Aşağıdaki class gerçek projelerde aynen kullanılan format:

```csharp
using MediatR;
using MyApp.Application.Dtos;
using MyApp.Common;

namespace MyApp.Application.Auth.Commands
{
    public class LoginCommand : IRequest<Result<LoginResultDto>>
    {
        public string Email { get; }
        public string Password { get; }

        public LoginCommand(LoginRequestDto dto)
        {
            Email = dto.Email;
            Password = dto.Password;
        }
    }
}
```

Basit ama **çok kritik** bir sınıf.

✨ Özet: LoginCommand tam olarak ne yapar?

Sorumluluk	Açıklama
Veri taşır	DTO'dan aldığı email & password bilgisini handler'a iletir
MediatR mesajıdır	Handler'a "login işlemi isteği" iletir
İş yapmaz	DB, token, doğrulama — hiçbirini yapmaz
Handler'ı tetikler	MediatR → ilgili handler'ı otomatik bulur
Pipeline'a girer	Validation, logging, caching gibi behavior'lardan geçer

---

Eğer istersen:

📌 **LoginCommandHandler'ın tam çalışma mantığını da adım adım anlatayım**
📌 Ya da tüm login modülünü (Controller + DTO + Command + Handler + Repo + JWT) bir bütün halinde

sana oluşturayım.

veritabanı okuma login işlemi nerede gerçeklşti

Login sırasında veritabanı okuma işlemi *tam olarak* "Application Handler'ın içinden, Infrastructure Repository'si kullanılarak" yapılır.

Aşağıda bunu *çok net ve adım adım* anlatıyorum.

✅ Login için veritabanı okuma işlemi hangi katmanda gerçekleşir?

🔥 Cevap: Application Layer'daki "LoginCommandHandler" içinde, Infrastructure katmanındaki "UserRepository" kullanılarak yapılır.

Bu çok kritik:

- **API katmanı** DB'ye dokunmaz
- **DTO** sadece veri taşır
- **Domain** business rule içerir
- **Infrastructure (Repository)** gerçek DB bağlantısını içerir
- **Application Handler** login akışını yönetir ve repository'i **sadece interface üzerinden** çağırır

---

📌 Akışın net şeması

```
scss

Controller → LoginCommand → LoginCommandHandler
    → IUserRepository (interface)
        → UserRepository (Infrastructure)
            → DbContext → SQL'e gider
```

Yani **login için veritabanı okuma = Application Handler + Repository**.

🧠 Neden böyle?

Çünkü:

- Application Layer iş akışını yönetir
- Infrastructure Layer veritabanıyla konuşur
- Application ile Infrastructure arasındaki bağlantı **IUserRepository** interface'idir

- Bu, Clean Architecture'ın çekirdeğidir

🧩 Şimdi tam kod örneğiyle gösterelim

🔻 1. Application Layer → LoginCommandHandler (asıl login burada yapılır)

```
csharp

public class LoginCommandHandler
    : IRequestHandler<LoginCommand, Result<LoginResultDto>>
{
    private readonly IUserRepository _userRepository;
    private readonly ITokenService _tokenService;

    public LoginCommandHandler(IUserRepository userRepository, ITokenService tokenService)
    {
        _userRepository = userRepository;
        _tokenService = tokenService;
    }

    public async Task<Result<LoginResultDto>> Handle(LoginCommand request, CancellationToken ct)
    {
        // 1) VERİTABANI OKUMA BURADA GERÇEKLEŞİR
        var user = await _userRepository.GetByEmailAsync(request.Email);

        if (user == null)
            return Result.Failure<LoginResultDto>("Kullanıcı bulunamadı");

        // Domain rule
        if (!user.VerifyPassword(request.Password))
            return Result.Failure<LoginResultDto>("Email veya şifre hatalı");

        var token = _tokenService.GenerateToken(user);

        return Result.Success(new LoginResultDto
        {
            Token = token,
            Name = user.Name
        });
    }
}
```

📌 Gördüğün gibi veritabanı okuma bu satırda gerçekleşiyor:

```
csharp

var user = await _userRepository.GetByEmailAsync(request.Email);
```

🔻 2. Domain Layer → IUserRepository (interface)

```
csharp

public interface IUserRepository
{
    Task<User> GetByEmailAsync(string email);
}
```

Application bu interface'i bilir, ama *nasıl çalıştığını bilmez*.

🔻 3. Infrastructure Layer → UserRepository (gerçek DB sorgusu burada)

```
csharp

public class UserRepository : IUserRepository
{
    private readonly AppDbContext _db;

    public UserRepository(AppDbContext db)
    {
        _db = db;
    }

    public async Task<User> GetByEmailAsync(string email)
    {
        return await _db.Users
                        .Where(x => x.Email == email)
                        .FirstOrDefaultAsync();
    }
}
```

📌 Burası gerçek EF Core sorgusudur ve **login sırasında veritabanından okuma işlemini yapan katman burasıdır.**

🔍 Özet Çizelgesi

İşlem	Katman
JSON'u almak	API
DTO oluşturmak	API
Command oluşturmak	API
Command'ı MediatR'a iletmek	API
Login akışını yönetmek	Application
Veritabanından kullanıcıyı okumak	**Infrastructure (Repository)**
Şifre doğrulamak	Domain
Token üretmek	Infrastructure / Application
Sonuç döndürmek	API

💬 Dilersen...

İstersen tüm bu login sürecini **sıfırdan çalışan bir proje** olacak şekilde (API + Application + Domain + Infrastructure klasörlü) kurup sana **tam bir ZIP** olarak çıkarayım.

İster misin?

vs code ile bu katmanları nasıl oluşturup sln yapısına ekleyebilirim

Aşağıdaki rehber, **VS Code** içinde **katmanlı .NET Core Clean Architecture çözümünü** sıfırdan nasıl kuracağını *adım adım ve uygulanabilir şekilde* anlatır.
Bu rehber sonunda:

✔️ `src/` altında 4 katman (API, Application, Domain, Infrastructure)
✔️ Bunların hepsini tek `.sln` altında toplama
✔️ Katmanlar arası doğru dependency ilişkisinin kurulması
✔️ VS Code içinde çalışır hâlde proje

elde etmiş olursun.

🎯 Hedef Çözüm Yapısı

```
css

MyProject.sln
src/
  MyProject.Api
  MyProject.Application
  MyProject.Domain
  MyProject.Infrastructure
```

✅ 1. Ana klasörü oluştur

Terminalde aç:

```
bash

mkdir MyProject
cd MyProject
mkdir src
cd src
```

✅ 2. Her katmanı oluştur (VS Code terminali)

1️⃣ Domain Layer

```
bash

dotnet new classlib -n MyProject.Domain
```

2️⃣ Application Layer

```
bash

dotnet new classlib -n MyProject.Application
```

3️⃣ Infrastructure Layer

```
bash

dotnet new classlib -n MyProject.Infrastructure
```

4️⃣ Api Layer (REST API)

```
bash

dotnet new webapi -n MyProject.Api
```

🎯 Şu an klasör yapın:

```
css

src/
  MyProject.Api/
  MyProject.Application/
  MyProject.Domain/
  MyProject.Infrastructure/
```

✅ 3. Solution (.sln) oluştur ve projeleri ekle

Önce üst klasöre çık:

```
bash

cd ..
```

Solution oluştur:

```
bash

dotnet new sln -n MyProject
```

Projeleri çözüme ekle:

```
bash

dotnet sln MyProject.sln add src/MyProject.Domain
dotnet sln MyProject.sln add src/MyProject.Application
dotnet sln MyProject.sln add src/MyProject.Infrastructure
dotnet sln MyProject.sln add src/MyProject.Api
```

✅ 4. Katmanlar arası bağımlılıkları ekle

Bu adım çok kritik. Clean Architecture kuralı:

- **Api → Application**
- **Application → Domain**
- **Infrastructure → Application + Domain**
- **Domain hiçbir şeye bağımlı değil**

Şimdi bunları terminalden ekleyelim.

Domain bağımlılık almaz (skip)

Application → Domain

```
bash

dotnet add src/MyProject.Application reference src/MyProject.Domain
```

Infrastructure → Application + Domain

```
bash

dotnet add src/MyProject.Infrastructure reference src/MyProject.Application
dotnet add src/MyProject.Infrastructure reference src/MyProject.Domain
```

Api → Application + Infrastructure

```
bash

dotnet add src/MyProject.Api reference src/MyProject.Application
dotnet add src/MyProject.Api reference src/MyProject.Infrastructure
```

🎯 Katman bağımlılık grafiği şimdi şöyle:

```
mathematica

MyProject.Api
   ↓ Application
   ↓ Infrastructure
Application → Domain
Infrastructure → Application + Domain
Domain → (hiçbiri)
```

Bu tamamen doğru Clean Architecture ilişkisidir.

✅ 5. VS Code içinde aç

Terminalden:

```
bash

code .
```

🧱 6. Klasör yapısını VS Code Explorer'da görürsün

```
css

MyProject.sln
src/
  MyProject.Api/
  MyProject.Application/
  MyProject.Domain/
  MyProject.Infrastructure/
```

✨ Bonus: Önerilen klasör iç yapısı (çok profesyonel)

📌 Domain

```
pgsql

Entities/
ValueObjects/
Enums/
Events/
Exceptions/
Services/ (domain services)
```

📌 Application

```


Interfaces/
Commands/
Queries/
Dtos/
Behaviors/
Mappings/
Validators/
```

📌 Infrastructure

```
pgsql

Persistence/
  AppDbContext.cs
  Repositories/
  Configurations/
Security/ (JWT)
```

```
Email/
Services/
```

📌 Api

```
Controllers/
Middlewares/
Configurations/
```

🚀 Son Adım: Build & Run

```
bash

dotnet build
cd src/MyProject.Api
dotnet run
```

API artık:

```
arduino

https://localhost:5001
```

altında çalışır.

💬 İstersen...

Bu yapıya göre **boş bir login modülü**, **repository yapısı**, **DbContext**, **JWT servis** veya **global response** yapısını da sıfırdan oluşturup sana verebilirim.

Hazır olarak tam çalışan bir örnek ister misin?