1. What is Layered Architecture?

Layered Architecture divides an application into multiple logical layers, each with a single responsibility and clearly defined roles. This design pattern is widely used in enterprise applications and ensures:

- **Separation of Concerns:** Each layer focuses on a specific responsibility.
- **Maintainability:** Each layer can be modified or replaced independently.
- **Testability:** Layers can be unit tested by mocking dependencies.
- **Scalability and Flexibility:** Layers can evolve or scale without affecting others.

2. Folder Structure Overview

```
MySolution.sln
├── MyApp.API                  # Presentation Layer
│   ├── Controllers            # API controllers
│   ├── Program.cs             # App startup, DI setup
│   ├── appsettings.json       # Configurations
│   └── DependencyInjection
│       └── APIInjector.cs     # API-layer DI registrations
│
├── MyApp.BLL                  # Business Logic Layer
│   ├── Interfaces             # Service interfaces (e.g., IUserService)
│   ├── Services               # Service implementations (e.g., UserService)
│   └── DependencyInjection
│       └── BLLInjector.cs     # BLL-layer DI registrations
│
├── MyApp.DAL                  # Data Access Layer
│   ├── Context                # EF Core DbContext
│   ├── Entities               # EF entities
│   ├── Repositories
│   │   ├── Interfaces
│   │   └── Implementations
│   ├── Migrations
│   └── DependencyInjection
│       └── DALInjector.cs     # DAL-layer DI registrations
│
└── MyApp.Shared               # Shared Layer
    ├── DTOs                   # Data Transfer Objects
    ├── Mappings               # AutoMapper Profiles
    └── Helpers                # Utility classes
```

3. Core Layers and Responsibilities

Layer	Responsibility	Typical Contents
API Layer (Presentation)	Expose endpoints, handle HTTP requests and responses	Controllers, routing, filters, middleware
Business Logic Layer (BLL)	Enforce business rules, validations, workflows	Service interfaces and implementations, DTO mapping
Data Access Layer (DAL)	Database CRUD operations, queries, persistence	DbContext, entity models, repository interfaces and implementations
Shared/Common	Shared models and helpers across layers	DTOs, AutoMapper profiles, constants, utilities

4. Layer Responsibilities and Interaction

API Layer

- **Purpose:** Entry point for HTTP clients.
- **Contains:** Only thin controllers.
- **Calls:** Business Logic Layer (not directly DAL).
- **Example:** UsersController calls IUserService.

Business Logic Layer (BLL)

- **Purpose:** Core business rules and workflows.
- **Contains:** Service interfaces and implementations.
- **Responsibilities:**
  - Orchestrate calls to repositories in DAL.
  - Map entities to DTOs and vice versa.
  - Validate business rules.
- **Example:** UserService implements IUserService, calls IUserRepository.

Data Access Layer (DAL)

- **Purpose:** Isolate all data access logic.
- **Contains:** DbContext, entity classes, repository interfaces and implementations.
- **Responsibilities:**
  - Perform CRUD operations.
  - Hide EF Core details from BLL.
- **Example:** UserRepository implements IUserRepository.

Shared Layer

- **Purpose:** Reusable components and models.
- **Contains:** DTOs, mapping profiles, helpers.
- **Example:** UserDto, UserMappingProfile.

5. Dependency Injection Setup

DALInjector.cs

```
public static class DALInjector
{
   public static IServiceCollection AddDALServices(this IServiceCollection services)
   {
      services.AddScoped<IUserRepository, UserRepository>();
      services.AddScoped<IOrderRepository, OrderRepository>();
      return services;
   }
}
```

BLLInjector.cs

```
public static class BLLInjector
{
   public static IServiceCollection AddBLLServices(this IServiceCollection services)
   {
      services.AddScoped<IUserService, UserService>();
      services.AddScoped<IOrderService, OrderService>();
      return services;
   }
}
```

Program.cs

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

builder.Services.AddDALServices();
builder.Services.AddBLLServices();

builder.Services.AddControllers();
builder.Services.AddSwaggerGen();
builder.Services.AddAutoMapper(typeof(UserMappingProfile).Assembly);

var app = builder.Build();

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run()
```

6. Layer Interaction Examples

Controller calls BLL service (recommended):

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
   private readonly IUserService _userService;
   public UsersController(IUserService userService) => _userService = userService;

   [HttpGet("{id}")]
   public async Task<IActionResult> GetUser(Guid id)
   {
      var userDto = await _userService.GetUserByIdAsync(id);
      if (userDto == null) return NotFound();
      return Ok(userDto);
   }
}
```

BLL calls DAL repository:

```
public class UserService : IUserService
{
   private readonly IUserRepository _userRepository;
   private readonly IMapper _mapper;

   public UserService(IUserRepository userRepository, IMapper mapper)
   {
      _userRepository = userRepository;
      _mapper = mapper;
   }

   public async Task<UserDto?> GetUserByIdAsync(Guid id)
   {
      var entity = await _userRepository.GetByIdAsync(id);
      return entity == null ? null : _mapper.Map<UserDto>(entity);
   }
}
```

7. Best Practices and Notes

- Keep controllers thin.
- Business logic should live in the BLL, not in controllers.
- DAL should be isolated for data persistence.
- Use interfaces to enable testability and loose coupling.
- Avoid injecting DbContext directly into controllers.
- Use DTOs for external contracts and never expose EF entities directly.

This layered approach ensures a clean, maintainable, and testable ASP.NET Core Web API that is easy to scale and extend.