Application Server layered Arch

* Presentation → presents data, accepts user inputs, communicates with

* Infrastructure layer → Appsettings.json
  - logging - caching - authentications - DbContext & Connection String
  - configuration management
  ↳ supports all layers

default variable.

* Layered arch and testability
  - each layer is independant → easier to isolate → test individual components
  - changes in one layer don't directly affect others
    ↳ More code maintainability and reduces regression.

* Best practice for managing communication between layers.
Dependancy inversion → use interfaces, to define contracts between layers, Avoid direct dependancies between layers to ensure loose coupling and flexibility.

well structured app server layer on preformance requests processed efficiently, manages flow of data, reduce redundancy, enable load balancing and scalability.

DTO

→ transfer only
→ promotes better layer boundries
→ used for core business logic
→ simple objects used to transfer data between parts of app or over network

more mapping code but increase complexity
→ increase preformance + security

DTO (data transfer object) → object, used for data transfers between software components,
↳ reduces method calls → bundles multiple data fields into one object
↳ Increases security (filters data returned). → No behaviour only attributes
↳ reduce amount of data sent especially on APIs

used for transfer — security ←

core logic
Domain models → contain both attributes and functionality
↳ like the ones we used to use before using DB.
↳ Represent real world entities (order, product, customer) contain attributes → ID, name ...
and business logic/rules → calculate total, check inventory ...
↳ Can combine data from multiple entities → only contain whats needed
Add sec — API doesn't need to get internal workings of domain model

Domain Model —mapping→ DTO

loosely coupled → achieved by reduced dependancy

* DTO Process:

- DTO used to transfer data between client + server
- API endpoint can send/receive DTO
- serialized to json

* Preformance increased → avoiding the transfer of large/unneeded data.
  ↳ Reduce payload size → improves network preformance and reduces amount of processing required on client side.

Common mistakes → Overuse, Complex mapping, too much data exposed.

Repository pattern vs Generic Repo

- Interface for each repo
- flexible
- less specific
  - ↳ BL can be applied
- Use: when a lot of BL
- Higher duplication
- Cleaner - more testable

- One interface for all CRUD operations. *Abstracts → supports testing
- Can have one repo for all
- less flexible
- less specific
- Use: when a lot of repitition in CRUD
- less duplication. *Reusability
- *Code reusability *Maintainability
- *DRY ✗NOT Flexible *Consistancy

Interface for each repo
↳ works with of DbContext
↳ This will interact with DB.

SOC
Encapsulation
higher level abstraction
↳ Interface.

*Repo → middle layer between DB and rest
- Unit of work. can encapsulate repo Pattern
  ↳ Abstracts data

Unit of work → coordinating multiple repos → all committed / rolled back
*encapsulation of actions (request) *Coordinates writing of changes
→ All business operations done as one unit
↳ Data consistency
*Can connect multiple repos.

Benifits → SOC, dec transaction management seperate from BL → cleaner → more maintainable
Transaction management, centralized control, consistency, data integrity

Cons → complex (especially with many repos), overhead, may be hard to scale (because of concurancy issues)

*Useful in applications with complex data operations across multiple entities/repos such as eCommerce/financial systems.

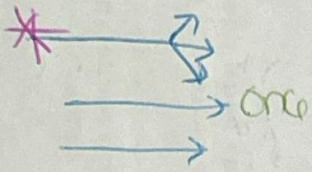
Dependancy Inversion Injection

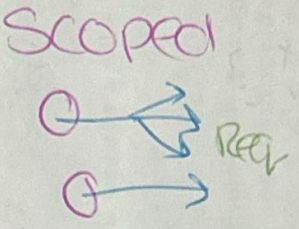
* High level modules should not depend on low level modules *

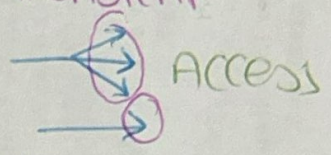
*One of the SOLID Principles

builder.Services.AddDbContext
↳ Always Scoped → all components of req executed together or rolled back → once per request → efficient and right weight
• Unit of work • Transaction • Best practice • Preformance + resource management

Singleton → resource management + concurency issues.







Injection types property, constructor, method. → passed through parameters.
↳ set through properties ↳ passed through constructor

Dependancy injection design pattern where Services and components (dependancies) are provided to a class rather than creating them.

Pros: loosely coupled, easy to swap implementation, testability, flexability,

Object lifecycle defines how long a service instance remains in memory

Dependancy injection code
Configure DI in Startup.cs (add Lifetime to services here)

Cons: Complexity, overuse.

Middleware and middleware pipeline → Adds overhead ↳ don't add too much ↳ slows down
↳ Software that is assembled into an application pipeline
↳ Process requests
↳ Can modify or short circuit

• middleware pipeline each req taken through each layer it can be processed, modified or passed on down the chain
↳ enables routing, logging, authentication, auth, error handling

* Setup in Startup.cs file

* Acts as a check point (for authen/author)

* Sequence is important bec it is order of execution

AppSettings.json → read at runtime.

*Stores application configuration (connection string, logging setting, API keys and any other configuable values)
↳ key value pair format
↳ Flexibility, SoC, maintainability ↳ clear, organized.

*Can access through startup.cs file

[Strongly typed → fields and properties known before runtime]

[User secrets → feature that stores sensitive data.
↳ for development only → Azure key vault / AWS secret manager

Restful API and HTTP protocol

easier to implement
↳ more scalable than SOAP API
long term
Stateless

• Representational state Transfer [idempotent does not create, change, no server state]
- Uses CRUD
↳ Get, Post, Put, Patch, Delete.

key principles:

- Stateless – all info must be filled in request
- Client-server arch – client and server are independant
↳ can change
- Uniform interface
- Cacheability
- layered system
- Code on demand

HTTP status code
200 – OK
201 – Created
400 – Bad request
404 – Not found
500 – Internal Server error

Idempotent
Get
PUT → does not change server state

NOT
POST

Content type header
↳ Type of data to be sent

API versioning → URL segmentation (param) / request readers.
/api/v1/resource

Error handling in REST API → return appropriate HTTP status code + error message

Swagger API Documentation

- Swashbuckle → writes your docs.
- Swagger config → Startup.cs
- Swagger generates documentation based on XML and annotations
- Swagger → framework for describing and documenting REST APIs
  - ↳ configured in Startup.cs
- Purpose: web interface for testing APIs
- Swagger annotation [HttpGet] [HttpPost]
  - ↳ control how they appear in swagger documentation
- POST [swagger API doc]
  - ↳ interactive, standardized format, code generation
    - ↳ client SDKs
- Securing → Authentication, limit access by roles, disabling swagger in production → add authorization middleware, IP whitelisting
- Risk (public API swagger docs) → sensitive info exposed (available API endpoints and authentication methods.
- *Can change swagger UI (modifying HTML, CSS, JS or in swagger)

API versioning

Common strategies

- ↳ URL path *Most used → configured in Startup.cs → supports backward compatibility
- ↳ Query parameter - affects readability and caching
- ↳ HTTP header - cleaner URL but hard on dev and consumers
- ↳ Custom media type

1.x.x. major not comp with prev

x.1.x new feature

x.x.1 patch

* Compatability issue, maintainance challenge

Depreciation → no longer supported

Routing in .Net

↳ function with steps.

→ Process of matching HTTP requests to controllers and actions ① this is determined using the URL
② HTTP method (get, post etc) ③ templates in application

Conventional routing

- route defined globally
  ↳ routeConfig / Startup.cs

*Map controllerRoute → custom route
- Map DefaultControllerRoute → default

Attribute routing

- Directly defined in controller using attributes
  ↳ [Route("api/[controller]")]
  *More flexible.
  *More control.

Route parameter validation [Route("api/products/{id:int}")]

Efficiency minimize no. of routes, avoid overly broad route patterns, use route constraints to make routing faster.

Model binding and validation

↳ Maps json data from HTTP request to C# model properties and objects → done in controller API ↳ enables BL

*Validation done after model binding