**Dependency Injection:**

Asp.net core has a built-in DI container.

Convention – populate the container in Startup.ConfigureServices method.

All the Services need to be registered in the same single method using ServiceDescriptor (this contains type of service, class that contains the implementation of this service, lifetime of this service)

If we use helper methods for the service registration, they take care of service registration. They have all the 3 details. We just called the extension method. The code within this extension method contains the service descriptor and in turn that contains all the details. (like EFs – default lifetime is scoped)

If we create our own services. Like UnitOfWork, we are instantiating manually in every file. We can instead depend on the DI container to resolve this dependency. Avoid using new keyword everywhere for UnitOfWork.

Here service descriptor would have the below properties:

* Type (always an Interface or an abstract class) -> IUnitOfWork
* Class Implementing -> UnitOfWork
* Service Lifetime ->

IServiceCollection – is the DI container.

IServiceProvider – contains the services (like Unit Of work, DB context)

Dependency -> Service A , Service C , component (here controller) dependent on the service. To instantiate the controller we need the Service A, Service C (like unit of work , DBContext), During run time, dependencies get resolved. Unless controller is not being instantiated, it won’t check for the dependency graph, if they have been registered or not.

3 lifetimes are available ->

Transient, SingleTon and Scoped

Transient – simplest lifetime -> if you are new to DI and core, use this

(note required to be thread safe, potentially less efficient)

Controller -> invokes CH -> invokes repository -> invokes DB. Lets assume all of these components have a dependency on Service A. If Service A was registered with Transient lifetime, each component will get a new instance of Service A. One for each of the dependent object. If we are not sure, if sharing state of the transient will cause an issue, then use Transient.

SingleTon -> onse single instance of the service is shared by all the components.

More performant, must be thread safe, recommended for functional stateless service.

If five simu;ltaneous request, 5 request , 5 controllers -> all compenents use the same instance, if the code is not thread safe and uses same instance, bugs incorporated which are extremen=ly difficult to pin point

Scoped -> A new instance of service is created, once per scope. Every HttpRequest is for the DI is like a new scope. Per HttpRequest, one new instance of the service A created for each request -> controller, CH, Repo.

We can create our own scope – custom scope – we can club just 2 components and not all in one scope.

HttpRequest is just a default scope.

Let start coding the right way without using the new keyword everywhere.

Avoid injecting DBCOntext directly. Instead use IUnitofWork. If you give DBContext to developers, they will have visibility to it.

Scoped – we are going with in this because in transient, Iunitofwork is being used multiple times => multiple IunitOfWork -> multiple DB connections -> then commits on one thread not in par with other other commits.

Singleton -> one users tx will get mixed up with other user’s tx. And updates weird.

A singleton service should not depend on scoped or transient because singleton lifetime is long, other services lifetime are shorter.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Tr | Sc | Si |
| tr | yes | yes | yes |
| sc | no | yes | yes |
| si | no | no | yes |

Captive dependency:

A service should not depend on another service with a shorter lifetime.

Issue -> sharing of non-thread

Objects live longer than required.

Principals of garbage collection-> if si object is holding on to sc, then even if sc object had to let go, its still there because si is holding on to it because its dependent on it.

ValidateScopeLifetime -> use this in the development environment. Costly validation. Check happens only in dev env, if you change the environment from development (in properties -> debug), to prod then error won’t come even though captive dependency.

If the same service is registered twice, then what happens -> registering 2 services of the same type (implementation differs, type same).

This will not result in an error but the most recent registration wins. Not a nice feature because depends on the order in which the services are registered because generally order in service registration doesn’t matter a lot but in this case it does. But both registration in the IServiceCollection.

Q.What if we want to use both ->>>>???????????? How does one use the earlier service.

A. In the dependent component (constructor) use Ienumerable<IUnitofWork> instead of IUnitofwork if Iunitofwork is implemented by multiple classes. And both the services are registered. If you use Iunitofwork here, the latest entry wins.

For every Add, there is a corresponding TryAdd. This will try to add the service if there is no service of the same type. Otherwise it ignores the second registration.

A much better way is to use TryAddEnumerable. It takes a collection of service descriptors. All the services in this Ienumerable get registered.

Try extension method for a better code quality. Like AddControllerwithviews.. we are creating registerservicesconfig in Infra folder.

Methods with “Add” are in configureServices.

“Use” in Configure method.

Manually resolving dependencies ???????? ☹ later next week he will teach this.

Use / Inject Iserviceprovider. Use Iserviceprovider.getservice / get requiredservices .

Implementation Factories -

Allow custom logic to create the object that needs to be used. Like IFilemanagement may be implemented by multiple classes but which class to use can put in func delegate.

Func delegate – good example he gave but couldn’t write everything.

Constructor Injection Logic – we have been doing that, all the parameters that are in the constructor should be in the DI or they should have a default value. And these constructors must be public.

If multiple constructors, core code will check for more feature rich constructor (multiple parameter). But all parameters should be in the DI or have dependencies resolved.

If same number of parameters, then it checks for the dependency graph, richer graph (i.e. more dependency)

Action Injection And view injection other types of injection mechanism.

If not all action in the controller use the service, then in those particular actions inject the Iunitofwork. But how does ASP.net understand that this prarameter is coming from the DI and not HTTPrequest. Use [FromService]. Used only in the action in MVC.

For views use @Inject in the cshtml pages.

Creating scopes Manually ???????/ ☹ he will teach later.

WEB API

Base class from which the WEB API inherits from is ControllerBase (both MVC, API) … very nice feature that only one base class.

Previously – Controller for MVC and ApiController for Web API.

Controller in core inherits from ControllerBase. (or the other way …please check)

ControllerBase won’t allow us to go ahead and let us return views from actions.

AddControllerwithviews will help both API and MVC.

AddController is for web apis.

Number of services registered would be significantly lesser. This is just a good practice, if apis only then go with addcontroller.

ApiController Attribute -> this is an optional parameter -> presence of this is good. -> things that are in industry standard. Enables API specific behaviors.

* Attribute routing requirement -> remember the routes pattern registration in global.asax.
  + Actions are inaccessible via conventional routes.
  + To enforce this we use Route attribute
  + Instead we can use the HTTPVerb with the route eg: [HttpGet(“/products”)]
  + Any route at the action level are relative to the route at the controller level.
  + Any (conventional) routes already registered in startup.cs would be used in the those controllers that do not have these routes on top.
  + Route Constraints. May be multiple. [Route(“users/{id:int:min(1)}”)]
  + Objective of these constraints is not validations. Only to help Asp.net to invoke the right action method.
  + Example is if 2 methods get user(int id) and get user (string name)
  + Both have some route constraints.
  + Now is you call get/200… then if 200 is not satisfied by either of them, user might expect bad request- 400 but instead gets 404 because page doesn’t exist.
  + We expect thinking that the validation didn’t work hence 400 but by design asp.net core doesn think of that as validation hence 404.
* Binding source parameter inference.
  + Below is ASP.net
  + Assume {c}/{a}/{id} someone does get c/a/2 and get c/a?id=2 and get c/a/2?id=3 and in body id=4.
  + Taking values from request and assigns to model I model binding. -first looks for rout param, then query string and then body.
  + But when you use APiController Attribute – it has its own logic to find the values from the input httprequest. Order is immaterial. In case of post request, and the input param is a complex object, then the complex object has to be looked into the body only only. But in case of primitive object, looks in the route and query string.
  + If the action is get and primitive input (int, string), first look in route and then in the query string but never in the body.
  + You could use FromQuery, FromRoute to change the default model binding inference behavior. What if you need to specify the order? No … 2 options can’t be provided.
* Automatic 400 response whenever any model validation error occurs.
  + Put data annotation in the model like required.
  + Then check modelstate == false then throw 400 --- this is automatically done in APIcontroller.
* Problem details for error status codes (400 or more) present only core 3.0 and above.
  + Based on RFC 7807 soec.
  + Some more detailed error information should be in the response.
  + Like if 50 fields on the request but something breaks then just 400 is not enough.
  + “Problem Details” object helps achieving the below JSON in the response.
  + This is an industry standard now.
  + The presence of the API controller automatically creates this object for us.

{

"type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",

"title": "One or more validation errors occurred.",

"status": 400,

"traceId": "|70b0c001-4e6955dd07999a46.",

"errors": {

"CustomerId": [

"The field CustomerId must be between 1 and 100."

]

}

}

* Work with Analyzers -> they do static code analysis while we are writing the code itself.
  + We will have to enable the analyzer.
  + If in get (id), if not order against that id then code should return 400 but our code doesn’t implement that.
  + So analyzer would flag saying that the code path is not implemented.
  + We will learn this later.

SWAGGER/ OpenAPI

* Language agnoustic specification for describing Web APIs.
* SWAGGER specification describes the capabilities of your API (like WSDL)
* SWAGGER UI provides information about the service from the spec.

Previously we had sOAP- WSDL standard but nothing for Rest API. That is being done by SWAGGER.

Helps testers, helps do documentation.

Lets see how to use this. Some component will analyse our code and will generate the specification.

Nuget package - swashbuckle.aspnetcore in MVC

<http://quartzsystems.com/downloads/core3/modelbinding.txt>

put the entire code in register method of method. For now remove the option auth.

Add it to the configureservices method.

Order is important while creating the pipeline.

Why useswagger added at the end of the Configure method. Research.

Use swagger() will give a JSON document and Use swagger UI will give you a nice UI.

app.UseSwagger().UseSwaggerUI(o => o.SwaggerEndpoint("/swagger/v1/swagger.json", "Heading I can customize"));

<https://localhost:44359/swagger/v1/swagger.json>

<https://localhost:44359/swagger/index.html> – The UI.

We can make this more detailed because it currently it just says 200 success. The definition is not 00% accurate. Our code can return so much more. We have not provided proper metadata. We haven’t documented the capability of our web api properly. It will start working properly when we start using analyser. Think of it as providing proper commenting. Lack of descriptive information shows just 200.