**Course description** :

"ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:

• Build web apps and services, IoT apps, and mobile backends.

• Deploy to the cloud or on-premises.

• Run on .NET Core or .NET Framework

The below topics will be covered in the Advanced .Net Core training:

1. Patterns for ASP.NET Core

2. Building Loosely Coupled Components

3. Web APIs

4. Model Binding

5. Filters

6. Tag Helpers

7. Middleware

8. Logging

9. Caching

10. Error Handling

11. Application Configuration

12. Identity and access control and security

13. State Management

14. Deployment and IIS Hosting"

SOLID principals

* Single responsibility
* Open/Closed
* Liskov Substitution
* Interface Segregation
* Dependency Inversion

Over engineering should not be done – improving the reliability of the software by implementing 1 or more of the above. If code is simpler by not introducing some of the above, still okay.

Patterns and how to write good code.

Single responsibility

Every software module (method or class) should have a focus on one single thing.

Small focused modules. Unit testing becomes easier for a focused module. (unit test is supposed to test a unit) Offloading the logic to multiple methods.

Logger object should be encapsulated, so that if tomorrow some changes to how logging happens, only that class/method needs a change.

Calling a web api – separate method – if way of calling changes.

Deserializing - today JSOn if tomorrow XML.

Open for extension – Closed for modification-

If you need to go back to the method over and over again – then more likely that you are introducing bugs.

Suppose new cases introduced, then same method you are changing again and again. Some cases removed. Modification of existing case.

Code that has been tested and deployed shouldn’t be changed again and again – reliability of software increased.

Can be achieved using Factory design pattern for both Single responsibility and open close principal – smaller independent business logic modules.

Liskov (LSP)

Deals with inheritance – if you have a method that accepts the base class. Then the same method should work with the child class as well. Subtypes must be substitutable for the base class.

Square(subtype) is a rectangle (base class).

Using IS and AS to do type checking in polymorphic code (not LSP)

Null check (LSP violation)

Presence of NotImplementedException (LSP violation)

Linq -> instead of FirstOrDefault use Single or First because only one record is going to be in the database.

Interface Segregation Principle

Clients should not be forced to depend on methods they do not use.

Prefer small, cohesive interfaces to larger and clunky ones.

Microsoft moved from Membership management to Iuser…

Dependency Inversion principal

Higher level modules should not depend on the ow level modules (both should depend on abstraction s

Abstraction should not depend on details

Details should depend on Abstraction

Data access layer – (lower level module) how data is stored in the DB

Middle tier (higher level module)– business logic

Application tier (low)

UI (presentation layer)

If a low ;level changes, then it should only impact that layer and not impact the higher tier.

UI changes then only App tier changes.

Database changes like SQL, Oracle, middler tier shiuld not need a change.

Public Interface Iorderrepository

{

Jobject getobjects (sqlparametercollection <- ADO.net object parameters)

}

Return type is JObject which is a low level detail and then the ADO.net SQL dependency .

List<order> getorders(dictionary<string, dynamic>, parameters);

Better way to represent things.

.net core used dependency injection that is in turn dependency inversion principal. Entre foundation is based on DI principal.

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

<http://quartzsystems.com/downloads/core3/day1.zip>

Application layer – on both the below – use entity object and interfaces – depends on infrastructure layer to use the repository implementation

Domain Model Layer – should not depend on any layer – business logic

Infrastructure Layer – depends on domain layer - EF

CQRS -------------------------------------------

.net core class libraries can be referenced in .net core only

.net framework in .net framework only

If interchangeably you want to reference it then use .net Standard.

.net core appsettings.json technically different from web.config but for now think of them as equivalent.

Asp.net normal works on event based – global.asax has those events based execution plan.

.net core has program.cs and startup.cs

.net core application requires a host (WPF, console, web based) in the program.cs

Appdomain in .net framework had to be created for the .net application to work.

.net core program cs when is creating the host, uses the startup initialization code from startup.cs. UseStartup method looks for configure services and configure methods. This is optional. This method. If you chose to comment the code, then write all the code in the program.cs itself. Use startup is a feature in itself and good to have in enterprise level application.

Configure services method is for dependency injection.

Configure is for the middleware pipelines and components.

HTTPHandlers and HTTPmodules -they would intercept the request – eg. Authentication, session,

.net core instead has middleware component and pipeline. Every request to the application goes through the middleware pipeline. You can create your own middleware component or use them out of the box provided by .net core.

Multiple pipelines can be available – use routing mechanism to route the request these pipelines.

He is using code first approach.

context.Database.EnsureCreated(); in the DBInitializer checks if the database is there then creates the tables and seeds the data in the tables.

DBContext has to be configured in the Configure Services. Order doesn’t matter for the services to be configured.

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection"));

Default Connection is the one present on the AppSettings.Json.

DataProtectionAPI, Azure Vault, etc. can be used to securely save our connection string. Right now in the JSON file.

var host = CreateHostBuilder(args).Build();

using (var scope = host.Services.CreateScope())

{

var services = scope.ServiceProvider;

var context = services.GetRequiredService<ContosoContext>();

DbInitializer.Initialize(context);

}

host.Run();

is to be written instead of -> CreateHostBuilder(args).Build().Run();

Right now it is => implies build and run but above piece of code actually initializes the DB and then go ahead and start running our application. Intialize method in the DBInitializer expects a ContosoContext. Build method has already configured the SQL object service.

 Repository Pattern:

Allows separate the logic that retrieves data and maps it to an entity model from the business logic that acts on it.

Enables the business logic to be agnostic to the data storage infrastructure.

Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection.

Repository is almost always used with another layer “unit of work”. – represent a series of activities carried out against a data store – Martin Fowler had said something about it. read it.

First repository layer and then UoW.

Individual Repository or a generic – separate dedicated class for each table in our DB – indiv. Creating independent repo is too much of repetitive work. Same code for insert update in all classes. In a real world he would create indiv repo for each table. He will explain why? Not his preferred approach. In a real world generic doesn’t work for everything.

Controller

UoW (DBContext)

Repository (DBSet)

DataAccess

DB

Middle 3 things are under Entity Framework.

Domain contains the contracts. Actual implementation goes in Infrastructure. GenericRepository (Infra) – uses generic parameters - Implements IRepository from Domain

Database should be exposed through UoW layer.

We can directly pass COntostContext Context to the generic repository constructor but everytime we use the repository, a new db connection is opened. While this is technically possible, we should instead have UoW layer in between.

unitOfWork.Context.Set<T>().AsEnumerable<T>();

IUow is not a contract for Domain. It can be in Infrastructure.

Uow – expose and dispose DbContext and provides this object to any other layer.

Generally it is DB layer -> Domain -> UI

Here Domain i.e. middle tier is independent.

At application layer we have the program.cs and Startup.cs which is going to initialize the DB so that can be used by both Infrastructure (which contains the EF) and the Domain.

Application

Infrastructure

Domain