Coding/API Creation Guidelines Document

(API/ Standard Coding/ Folder Structure)

Introduction:

This API creation guidelines document will explain the details of the following items:

- **Controllers**
- **DTO/ Models**
- **Services**
- **Repositories**
- **DB Entities**
- **Others**

Controllers:

1. All **Controller** should inherit from **XrmControllerBase/ XrmGetControllerBase/ XrmPostControllerBase** class, and implement its own **interface (should be internal)** if API endpoint is extended outside of what are there in **XrmControllerBase/ XrmGetControllerBase/ XrmPostControllerBase** class.
2. All controller **interface** should be internal.
3. All API should have **XML comment**.
4. All API should have Route attribute with API name.
5. All Controller should have Authorize attribute, if any API required anonymous access then apply AllowAnonymous attribute at API level.
6. All Controller API should have ApiVersion attribute as [ApiVersion("1.0")]
7. All API should have Custom SwaggerResponse Attribute [Possible API Response]
8. All API should be async and have ValueTask<IActionResult> as return type.
9. All API should have **HttpMethod** defined.
10. All API should return with OkObjectResult if success
11. All API should return with BadRequestObjectResult for other responses
12. All Data fetching APIs should be HttpGet methods.
13. All sensitive data fetch APIs should be HttpPost methods.
14. All Data creation APIs should be HttpPost methods.
15. All Data Modification APIs should be HttpPut methods.
16. All Data Deletion APIs should be HttpDelete methods.
17. All APIs that doesn't need Response body should be created as HttpHead method API.
18. All Controller should have the route "api/v{version:apiVersion}/{ControllerName}". {ControllerName} should not be the placeholder [Controller].
19. All **Controller's Interface** should be created inside **Interfaces>V1** folder
20. All controllers should be created inside **Controllers>V1** folder

21. All custom Response Types should have prefix "**Xrm**" and postfix "**Response**" in its name. e.g. XrmCustomTypeResponse
22. All Data Fetch APIs should implement **Paginations**.
23. Return only Ok and BadRequest from API wrapping the custom **ResponseTypes**.
24. Every API urls in **Gateway** should have a rate limiter.
25. API **aggregator** should be created only if requested from **Front End.**
26. API description should be properly available in **Swagger UI**. All API should have **XML comments** with proper Parameters comment and return type. And also, API should be marked with proper Possible Status Code **SwaggerResponse** Code.

```
[ApiController]
[Route("api/v{version:apiVersion}/AccessLogs")]
[ApiVersion("1.0")]                                   6
[Authorize]                                           5
0 references
public class AccessLogsController : ControllerBase, IPageAccessLog   1
{
    /// <summary>
    /// API to log Page Access from Frontend UI
    /// </summary>
    /// <param name="pageTitle">Title of the Page</param>
    /// <returns></returns>                           3
    [HttpPost]
    [Route("PageAccess")]                             4
7   [XrmSwaggerSuccess]                               8
    [AllowAnonymous]                                  5
    1 reference
    public async ValueTask<IActionResult> PageAccessLogAsync([FromQuery]string pageTitle)
    {
        return Ok(new XrmSuccessResponse());          21
    }
}
```

```
internal interface IPageAccessLog
{
        2
}
```

```
[HttpGet]
[Route("GetCultureInfoTestAsync")]
[XrmSwaggerSuccess]
0 references
public async ValueTask<IActionResult> GetCultureInfoTestAsync(int id)
{
    if (id == 0) return new BadRequestObjectResult("Invalid Request");
    var result = await _cultureInfoService.GetCultureInfoByIdAsync(id);
    if (result.succeeded)
    {
        return new OkObjectResult(result);   10
    }
    return new BadRequestObjectResult(result);   11
}
```

```
[HttpPost]   14   9
0 references
public async ValueTask<IActionResult> AddCultureInfoTestAsync()
{
    // code here
}
```

```
[HttpGet]   12
0 references
public async ValueTask<IActionResult> GetCultureInfoAsync()
{
    // code here
}
```

```
[HttpPut]   15
0 references
public async ValueTask<IActionResult> UpdateCultureInfoAsync()
{
    // code here
}
```

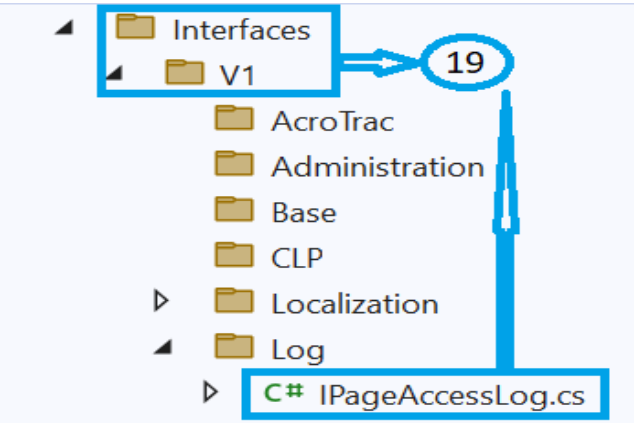
```
[HttpDelete]   16
0 references
public async ValueTask<IActionResult> DeleteCultureInfoAsync(int id)
{
    // code here
}
```

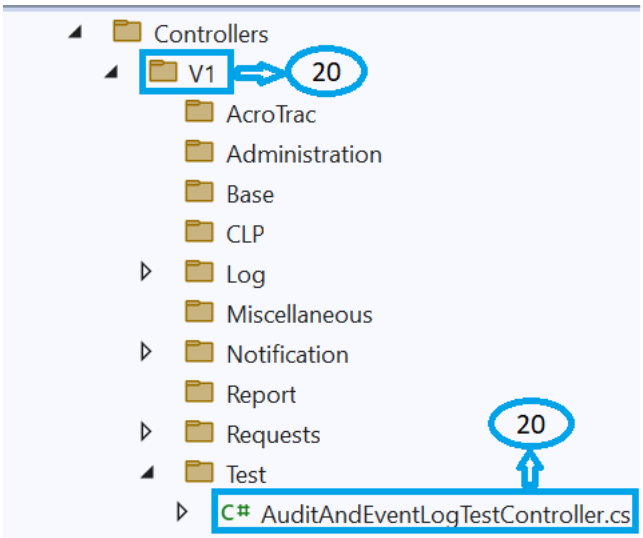
```
/// <summary>
/// Access Logs
/// </summary>
[ApiController]
[Route("api/v{version:apiVersion}/AccessLogs")]
[ApiVersion("1.0")]
0 references
public class AccessLogsController : ControllerBase, IPageAccessLog
{

}
```

18

19

20



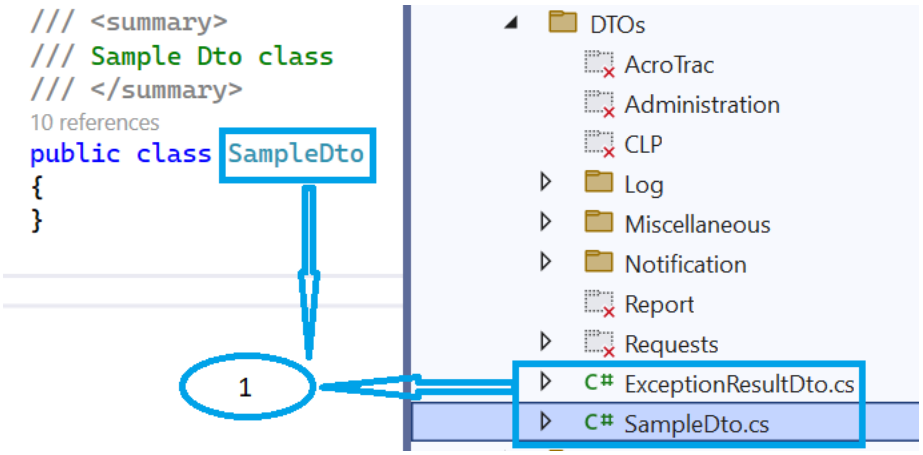


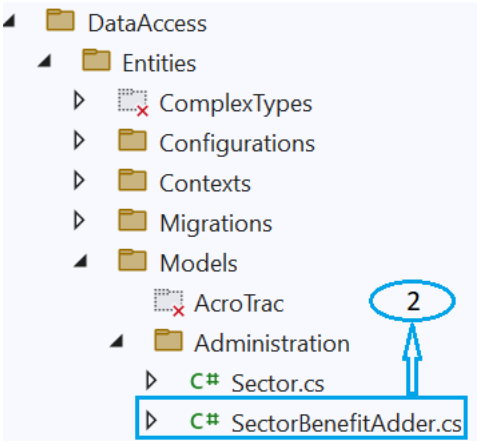
```
public async ValueTask<IActionResult> GetAllAccessLogAsync([FromQuery] string accessLogType, SamplePaginationDto paginationDto)
{
    if (paginationDto != null && (paginationDto.StartIndex < 0 || paginationDto.PageSize < 0))
        return BadRequest(new XrmErrorResponse("Invalid Pagination"));

    var result = await _logService.Test(accessLogType, paginationDto: paginationDto);
    if (result.Succeeded)
        return Ok(result);

    return BadRequest(result);
}
```

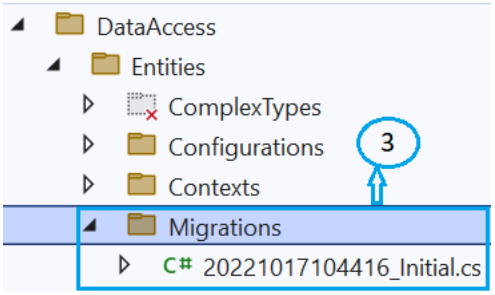
21

Dtos/Models:

1. All **DTO** classes should have postfix "**Dto**"
2. **All Models nullable and non-nullable should be reflected in model class also along with Fluent API. Example: if a string property is nullable then we should write "string?" in model class and in Fluent API "IsRequired(false)". Do not mismatch nullable type i.e. defining nullable in model class and make it non-nullable with Fluent API or vice versa.**
3. All **Models** should be created inside **DataAccess > Enttities > Models**
4. All **Models's Migration** file should be created inside **DataAccess > Entities > Migrations**
5. All **Model** class should disable **XML comment** warning with **"#pragma warning disable CS8618, 1591**" just below namespace line.
6. All Master data seeding for Model should be created inside **Seeder** folder.
7. All **Models** should inherit Entity abstract class. Entity Abstract class has Id and Ukey property.
8. All AutoMapper mapping configurations for Module specific Models and DTOs should be done from the specific module folder inside ObjectMapper.
9. All **DTO** data integrity validations should be done inside **BusinessLogic's** Service class
10. All **Model** data integrity validations should be done inside **DataAccess's** Repositories class
11. All DTO class properties should have proper XML Comment.
12. If DTO class property is different from Model's property then proper mapping should be done in AutoMapper with "ForMember" clause.

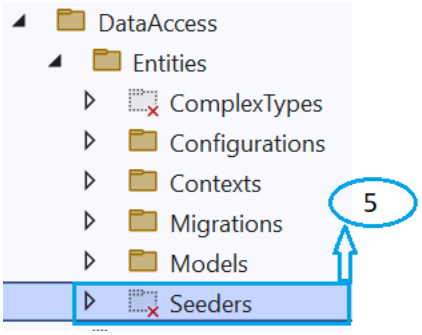

/// <summary>
/// Sample Dto class
/// </summary>
10 references
public class SampleDto
{
}
1
DTOs
AcroTrac
Administration
CLP
Log
Miscellaneous
Notification
Report
Requests
ExceptionResultDto.cs
SampleDto.cs


DataAccess
Entities
ComplexTypes
Configurations
Contexts
Migrations
Models
AcroTrac
2
Administration
Sector.cs
SectorBenefitAdder.cs


DataAccess
Entities
ComplexTypes
Configurations
3
Contexts
Migrations
20221017104416_Initial.cs

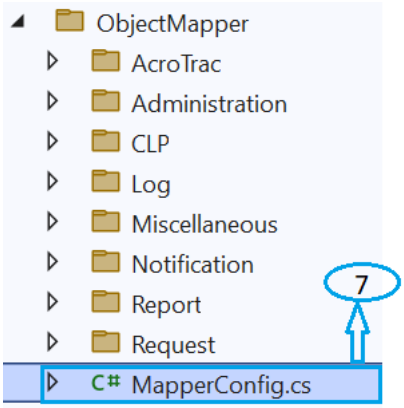
```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
27 references
public class Sector : EntityWithAudit
{
    4 references
    public string SectorName { get; set; }
    1 reference
    public string Address1 { get; set; }
    1 reference
    public string Address2 { get; set; }
    1 reference
    public string City { get; set; }
    1 reference
    public string State { get; set; }
    1 reference
    public string PostalCode { get; set; }
    0 references
    public bool IsBenefitAdder { get; set; }
    3 references
    public virtual ICollection<SectorBenefitAdder> SectorBenefitAdders { get; set; }
}
#pragma warning restore CS1591 // Missing XML comment for publicly visible type or member
```

4




```
public class SectorBenefitAdder : Entity
{

}
```

6



Services:

1. All service should implement its own **interface**.
2. All Services should handle error with **try catch** block.
3. All Services should return an instance of XrmInternalServerErrorResponse from **catch** block and provide exception as parameter.
4. All services should have return type as **Xrm Custom ResponseTypes**. If multiple response types are required from within the same service, then use parent ResponseBase class as return type.
5. Any services which will be/ could be used in multiple projects should be created inside Services folder in **CommonUtilities** project.
6. All services for specific modules should be created inside Services of **BusinessLogic** folder
7. All Interfaces for specific module services should be created inside Interfaces of **BusinessLogic** folder
8. All **classes (including Controller), interface, methods, properties, enum, struct** should have XML comment properly defined with parameter description except for **Entity Model** class, **Migration** class file.

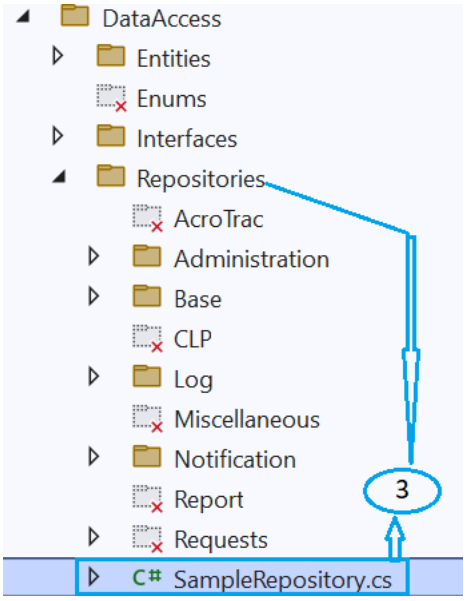
```
///<inheritdoc/>
1 reference
public class LogConfigService : ILogConfigService
{

    ///<inheritdoc/>
    2 references
    public async ValueTask<ResponseBase> AddAsync(LogConfigDto logConfigDto)
    {
        try
        {
            return new XrmSuccessResponse();
        }
        catch (Exception ex)
        {
            return new XrmInternalServerErrorResponse(ex: ex);
        }
    }
```

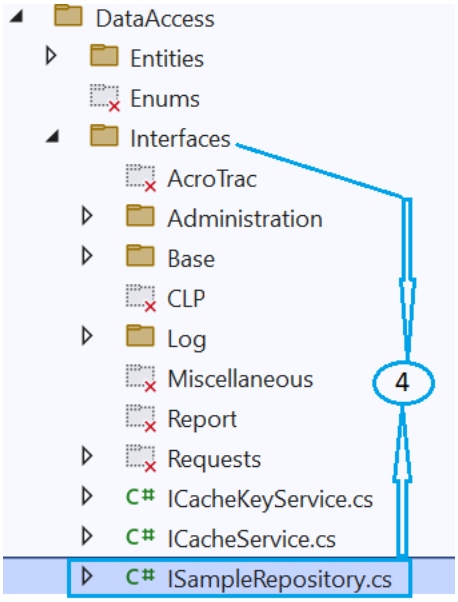
Repositories:

1. All repository should implement its own interface.
2. All repo should implement generic IRepository.
3. All **Repositories** should be created inside Repositories folder of **DataAccess** folder.
4. All **Repository's Interfaces** should be created inside Interfaces folder of **DataAccess** folder
5. All CRUD operations should be done through **Generic Repository**.

```
/// <summary>
/// Definition of all the methods of access log repository
/// </summary>
1 reference
public class AccessLogRepository : Repository<AccessLog>, IAccessLogRepository
{

}
```

```
/// <inheritdoc/>
7 references
public class Repository<T> : IRepository<T> where T : class
{

}
```





DB Entities:

1. All **DbContext** related changes should be done inside the specific module's **DbContext** file located inside **DataAccess > Entities > Contexts**
2. All Module related **Model's** Metadata configuration should be done from inside the specific module configuration folder **DataAccess > Entities > Configurations > [Module Folder]**
3. All complex return types of Stored Procedure should be created inside **DataAccess > Entities > ComplexTypes** folder.
4. No instance of **DbContext** will be created outside Repositories classes.

5. All **Stored Procedure, Views, Functions,** or any other Database objects should be created through **Entity Framework Migration** file. Direct creation of database object through **SQL Management Studio** is **NOT** allowed.
6. Deletion of **MigrationHistory** data is prohibited.
7. Never access **database/ DbContext** from the **BusinessLayer/ Service Layer**. Database manipulation should only be done at **Repository Layer**.

Migration:

1. Before running add-migration command, we should always checkout migration snapshot file (e.g. **XRMDbContextModelSnapshot.cs** for XRMDbCOntext). THIS IS MANDATORY.
2. Always make sure to provide Migration file path in **add-migration** command along with dbcontext.
   E.g.
   ```
   add-migration <MigrationFileName> -Context XrmDbContext -o
   DataAccess/Entities/Migrations
   ```
3. Migration File Name should be kept as short as possible. Avoid giving long migration file name.
4. If running migration on "EmailService", "LocalizationService", & "LoggingService" then make sure that update-database command has the database connection string attribute (-Connection).
   Example:
   ```
   update-database -Context AuthDbContext -Connection
   "Server=172.16.0.18\\SQL2017,1434; Database=XRMv2_Common;
   Trusted_Connection=True; MultipleActiveResultSets=True;
   TrustServerCertificate=True;"
   ```
5. If migration needs to be rollback then follow the below steps:
   a.

If you have DateTime field (like CreatedOn) that needs to be added while writing Seeder file then do not use DateTime.Now to set the value instead give a hard coded value like "new DateTime(2023, 1, 15, 0, 0, 0)". This will prevent your seeder from being included on other's migration file.

<u>Other Guidelines:</u>

1. All internal class should have **protected** **constructor**.
2. All private variable initialized in the **constructor** should be marked as readonly.
3. All classes that are not to be used outside the assembly should be marked as **internal**.
4. All methods that are not to be used outside the class by its instance should be marked as **private**.
5. All **internal** methods should be marked as **private**.
6. All **CS files** should not contain any unused namespace.
7. All **CS files** should not have Commented Code unless approved by **Leads**.
8. All framework-based files should never be modified without approval from **Framework Team/ Technical Managers**.
9. All framework-based files are closed for modification.

10. No **NuGet Packages** or **.NET Framework** should be upgraded or installed without approval from **Technical Managers**.
11. **Bin** and **Obj** folder should never be part of **Source Control** and should never be checked in.
12. No **Check-in** should be done without Comment and Associating **WorkItem (Tasks/ User Stories)**
13. Project References has already been added as per required. If new reference is needed, prior approval needs to be taken from **Technical Managers.**
14. Proper Unit Testing should be done through **Postman**.
15. All **IDE** should have **SonarLint** extension installed.
16. All warnings in each file should be address before **check-in**. If not able to resolved from developer's end should escalate to **Technical Lead/ Technical Manager/ COE Team** in the same order.
17. All naming conventions should be in **Pascal** case.
18. Use **PascalCasing** for class names and method names.
19. Use **Camel Casing** for local variables and method arguments.
20. All **Extension** methods/class/ file should be created inside **Extensions** folder
21. All **Extension** class/ file name should have postfix "**Extension**".
22. All **Filters** method should be created inside **Filters** folder
23. All **Enums** should be created inside **Enums** folder
24. All **Middleware** should be created inside **Middlewares** folder
25. All **Middleware** class/ file name should have postfix "**Middleware**".
26. Any new Project should be created inside **SRC** folder.
27. Any **Dependency Injection** of classes should be done from specific module DI file located inside the **DIRegistration** Folder.
28. All validators for specific **service/module** should be created inside validators of **BusinessLogic** folder
29. All Domain related logics and requirements should be done inside **BusinessLogic's** Service class.
30. All **Security Checklist** should be followed.
31. Used object should be disposed if applicable, It can be handled using implementation of IDisposable and used as "**Using**".
32. After any **check-in** done, developer must validate from peers after get latest to ensure there is no project compilation error. It often happens when check-in done, developer who **checked-in** the code everything work fine to there system, but it generate errors to others after getting latest file due to partial check-in or missing references, etc.
33. All Date format for model properties should be in datetime2 data type.
34. All String format for model properties should be in nvarchar data type.
35. A class should have only one responsibility.
36. A method should have only one responsibility.
37. All Entity Configuration class for **Fluent API** should be created inside "**Configuration**" of "**Entities**" folder within **DataAccessLayer**.
38. All Entity Configuration class/ file name for **Fluent API** should have postfix "**Metadata**".
39. Do not write comments for every line of code and every variable declared.
40. Use **//** or **///** for comments. Avoid using **/* … */**
41. If you have to use some complex or weird logic for any reason, document it very well with sufficient comments.

42. Avoid using else conditions.

```
//Bad
public void Test(Test test) {
   if (test == null) {
      throw new ArgumentNullException(nameof(test));
   } else {
      //Do something
   }
}

//Good
public void Test(Test test) {
   if (test == null)
      throw new ArgumentNullException(nameof(test));

    //Do something
}

//Bad
public Result Test(SomeEnum someEnum) {
   if (someEnum == SomeEnum.A) {
      //Do something
   } else {
      //Do the Other thing
   }
}

//Good
public Result Test(SomeEnum someEnum) {
   return someEnum switch {
      SomeEnum.A => DoSomething(),
      SomeEnum.B => DoTheOtherThing(),
      _ => throw new NotImplementedException()
   };

   Result DoSomething() { }
   Result DoTheOtherThing() { }
}
```

43. Use **LINQ expression** everyever possible for querying **Collection (List/ Collection/ IEnumerable)**
44. A line of code should not exceed half the screen. This makes it easier to read and you can have two files open in a split-screen, without missing some code.

45. Global **Usings** and **File-Scoped Namespaces**. In **C# 10** you now have the possibility to use file-scoped namespaces and global usings.
46. Do not use **Hungarian** notation or any other type of identification in identifiers:
    // Correct
    int counter;
    string name;

    // Avoid
    int **i**Counter;
    string **str**Name;

47. Do not use **Screaming Caps** for constants or readonly variables:
    // Correct
    public static const string ShippingType = "DropShip";

    // Avoid
    public static const string **SHIPPINGTYPE** = "DropShip";

48. Avoid using **Abbreviations** except for commonly used names such as **Id, Ukey, Xrm, Uri, Xml**.
    // Correct
    UserGroup userGroup;
    Assignment employeeAssignment;

    // Avoid
    UserGroup usrGrp;
    Assignment empAssignment;

49. Do not use **Underscores** in identifiers.
50. Use predefined type names instead of system type names like **Int16, Single, UInt64**, etc. Consistent with the **Microsoft's .NET Framework**. It makes code more natural to read.
    // Correct
    string firstName;
    int lastIndex;
    bool isSaved;

    // Avoid
    String firstName;
    Int32 lastIndex;
    Boolean isSaved;

51. Use implicit type var for local variable declarations. Exception: **primitive types (int, string, double, etc)** use predefined names.
    var stream = File.Create(path);
    var customers = new Dictionary();

```
// Exceptions
int index = 100;
string timeSheet;
bool isCompleted;
```

52. **Prefix** interfaces with the letter **I**. Interface names are noun (phrases) or adjectives.
53. Do name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.
54. Do vertically align curly brackets.

```
// Correct
class Program
{
   static void Main(string[] args)
   {
   }
}
```

55. Do declare all member variables at the top of a class, with static variables at the very top.

```
// Correct
public class Account
{
   public static string BankName;
   public static decimal Reserves;

   public string Number {get; set;}
   public DateTime DateOpened {get; set;}
   public DateTime DateClosed {get; set;}
   public decimal Balance {get; set;}

   // Constructor
   public Account()
   {
      // ...
   }
}
```

56. Do not use suffix enum names with **Enum**.
57. Always declare the variables as close as possible to their use.
58. Always separate the methods, different sections of program by one space.
59. Write only one statement per line.
60. Write only one declaration per line.
61. Use one of the concise forms of object instantiation, as shown in the following declaration.

```
ExampleClass instance2 = new();
```

Or

```
return new ExampleClass();
```

62. Do not use variable names that resemble keywords or existing **.NET Framework** classes.
63. Prefix boolean variables, properties and methods with **"is"** or similar prefixes.
64. Method name should tell what it does. Do not use mis-leading names. If the method name is obvious, there is no need of documentation explaining what the method does.

    Good:

    ```
    void SavePhoneNumber ( string phoneNumber )
    {
        // Save the phone number.
    }
    ```

    Not Good:

    ```
    // This method will save the phone number.
    void SaveDetails ( string phoneNumber )
    {
        // Save the phone number.
    }
    ```

65. A method should do only **'one job'**. Do not combine more than one job in a single method, even if those jobs are very small.

    Good:

    ```
    // Save the address.
    SaveAddress (  address );

    // Send an email to the supervisor to inform that the address is updated.
    SendEmail ( address, email );

    void SaveAddress ( string address )
    {
        // Save the address.
        // ...
    }

    void SendEmail ( string address, string email )
    {
        // Send an email to inform the supervisor that the address is changed.
        // ...
    }
    ```

    Not Good:

```
        // Save address and send an email to the supervisor to inform that
// the address is updated.
        SaveAddress ( address, email );

        void SaveAddress ( string address, string email )
        {
                // Job 1.
                // Save the address.
                // ...

                // Job 2.
                // Send an email to inform the supervisor that the address is changed.
                // ...
        }
```

66. Do not hardcode numbers. Use constants instead. Declare constant in the top of the file and use it in your code. However, using constants are also not recommended. You should use the constants in the config file or database so that you can change it later. Declare them as constants only if you are sure this value will never need to be changed. Use Enum wherever required. Do not use numbers or strings to indicate discrete values.

Good:

```
        enum MailType
        {
                Html,
                PlainText,
                Attachment
        }

        void SendMail (string message, MailType mailType)
        {
                switch ( mailType )
                {
                        case MailType.Html:
                                // Do something
                                break;
                        case MailType.PlainText:
                                // Do something
                                break;
                        case MailType.Attachment:
                                // Do something
                                break;
                        default:
                                // Do something
                                break;
```

```
        }
    }
```

Not Good:

```
    void SendMail (string message, string mailType)
    {
        switch ( mailType )
        {
            case "Html":
                // Do something
                break;
            case "PlainText":
                // Do something
                break;
            case "Attachment":
                // Do something
                break;
            default:
                // Do something
                break;
        }
    }
```

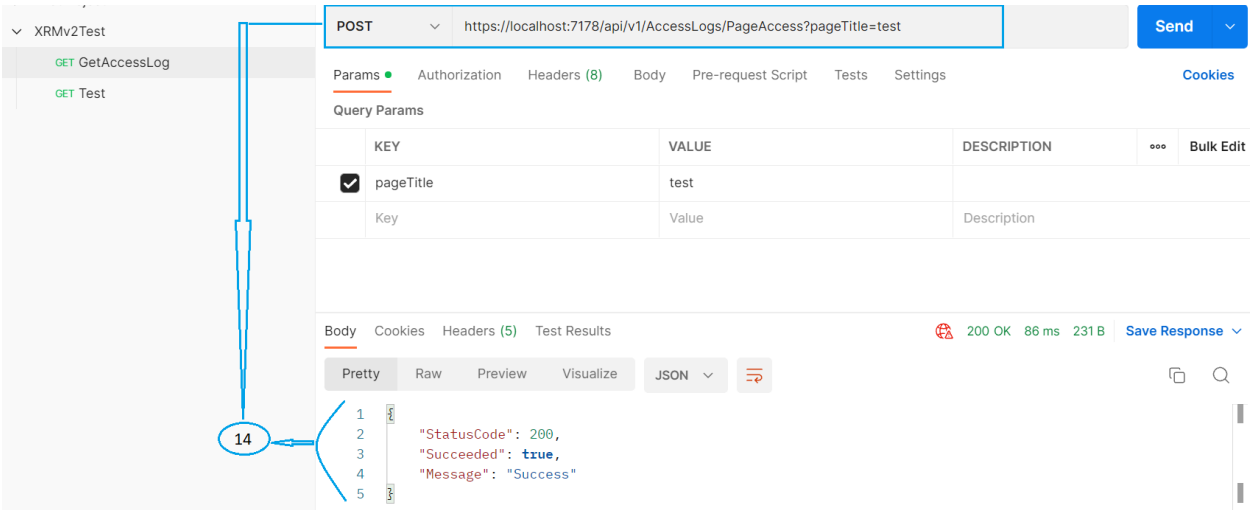
67. Never hardcode a path or drive name in code. Get the application path programmatically and use relative path.
68. Do not have more than one class in a single file.
69. Avoid writing very long methods. A method should typically have **1~30 lines of code**. If a method has more than **30 lines of code**, you must consider re factoring into separate methods.
70. Avoid having very large files. If a single file has more than **1000 lines of code**, it is a good candidate for refactoring. Split them logically into two or more classes.
71. Avoid passing too many parameters to a method. If you have more than **2~5 parameters**, it is a good candidate to define a class or structure.
72. If you have a method returning a collection, return an empty collection instead of NULL, if you have no data to return.
73. Use StringBuilder class instead of String when you must manipulate string objects in a loop.
74. Wherever possible, catch only the specific exception, not generic exception.
75. When you re throw an exception, use the throw statement without specifying the original exception. This way, the original call stack is preserved.
76. All sensitive data should be using Encryption & Decryption.
77. All API should use https
78. A non-Nullable bool or int should not have default value in metadata. If wished to have default value true/false for bool field and any numeric value for int then that field should be nullable.

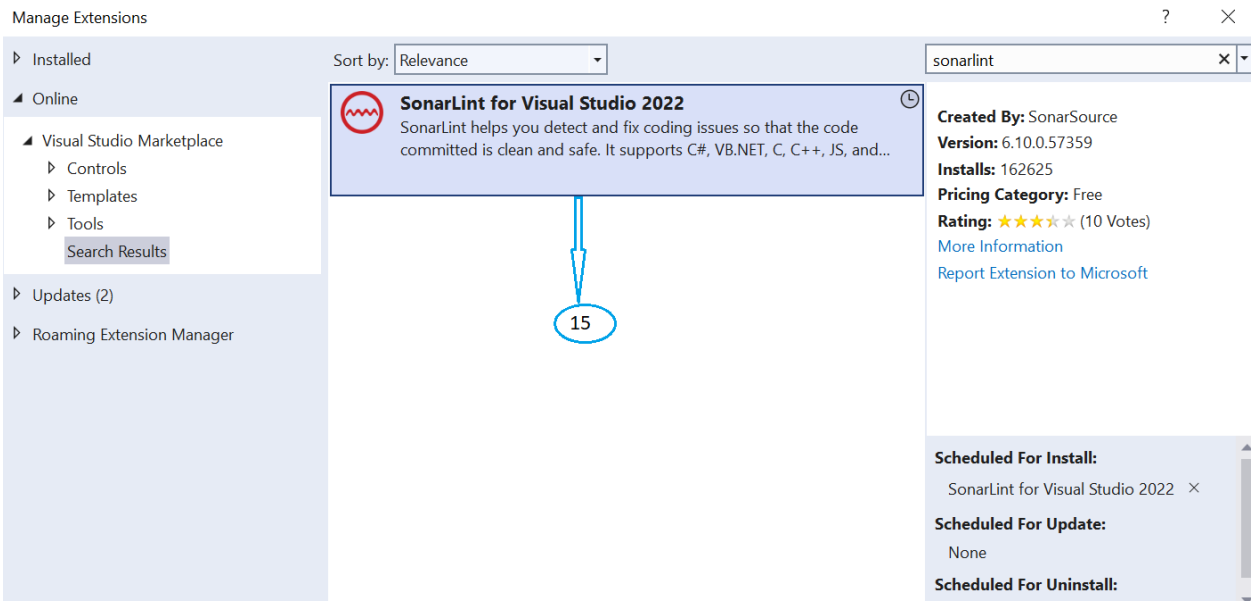
79. Do not add unnecessary two way Navigation property while creating model. Always go for One way unless you are sure that you need ICollection navigation property from the parent table.
80. Always check if your navigation property has circular reference/link. your navigation property from where it is generating should not be linked back from the parent table in any way through other navigation property. E.g. ReqLibrary to LaborCategory to JobCategory to ReqLibrary.
81. If circular reference is unavoidable then always have OnDelete(DeletionBehaviour.NoAction) while defining relationship in metadata file. DeletionBehaviour.Cascade should not be used in such scenarios.
82.

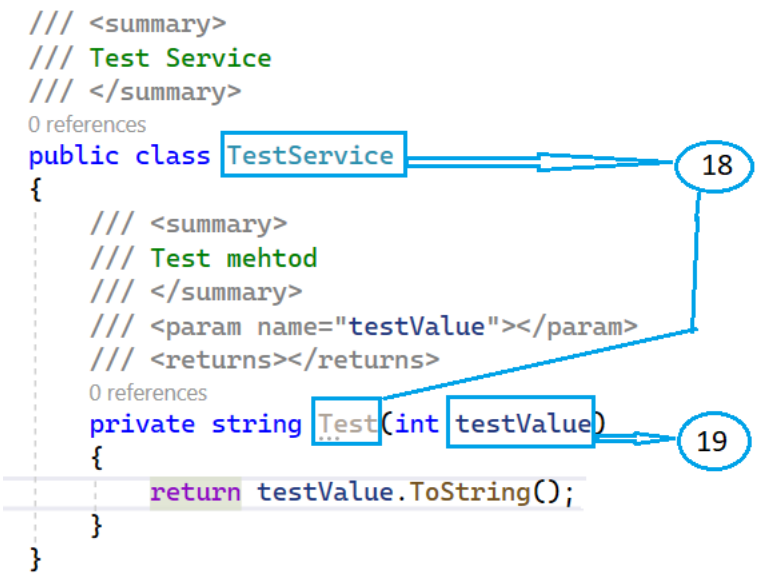
```
internal class AcroTracMetadata                                   1   3
{
    0 references
    protected AcroTracMetadata(){}

    /// <summary>
    /// Configure Acrotrac module related tables.
    /// </summary>
    /// <param name="modelBuilder">An instance of ModelBuilder</param>
    1 reference
    public static void Configure(ModelBuilder modelBuilder)
    {
        // Method intentionally left empty.
    }
    0 references                                                  4   5
    private string test(string timeInterval)
    {
        return timeInterval;
    }
}
```

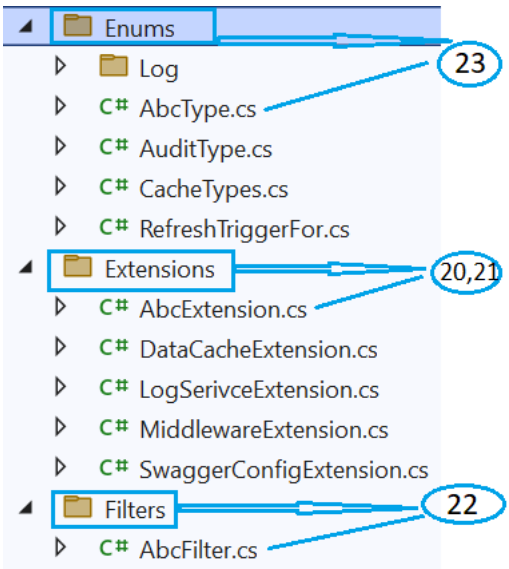
```
public class TestController : ControllerBase
{
    private readonly ISampleService _sampleService;
    0 references
    public TestController(ISampleService sampleService)
    {
        _sampleService = sampleService;
    }
    2
```

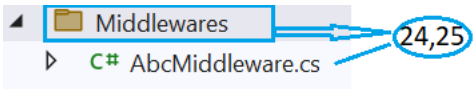
```
using APIService.DataAccess.Entities.Models.Base;
using System.Text;                                                    6
namespace APIService.DataAccess.Entities.Models.Administration
{
#pragma warning disable CS8618, 1591 // Missing XML comment for publicly visible type or member
    27 references
    public class Sector : EntityWithAudit                             18
    {
        //public string Id { get; set; }                              7
        4 references
        public string SectorName { get; set; }
        1 reference
        public string Address1 { get; set; }
        1 reference
        public string Address2 { get; set; }
        1 reference
        public string City { get; set; }
        1 reference
        public string State { get; set; }
        1 reference
        public string PostalCode { get; set; }
        0 references
        public bool IsBenefitAdder { get; set; }
        3 references
        public virtual ICollection<SectorBenefitAdder> SectorBenefitAdders { get; set; }
    }
#pragma warning restore CS8618,1591 // Missing XML comment for publicly visible type or member
}
```

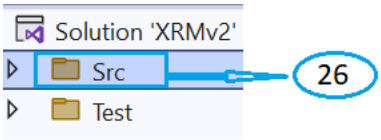



Manage Extensions
Installed
Online
Visual Studio Marketplace
Controls
Templates
Tools
Search Results
Updates (2)
Roaming Extension Manager
Sort by: Relevance
sonarlint
SonarLint for Visual Studio 2022
SonarLint helps you detect and fix coding issues so that the code committed is clean and safe. It supports C#, VB.NET, C, C++, JS, and...
15
Created By: SonarSource
Version: 6.10.0.57359
Installs: 162625
Pricing Category: Free
Rating: (10 Votes)
More Information
Report Extension to Microsoft
Scheduled For Install:
SonarLint for Visual Studio 2022
Scheduled For Update:
None
Scheduled For Uninstall:

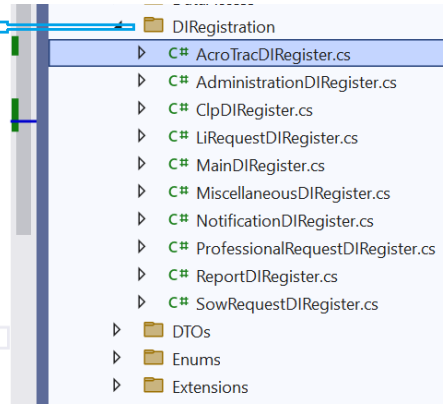

/// <summary>
/// Test Service
/// </summary>
0 references
public class TestService
18
{
/// <summary>
/// Test mehtod
/// </summary>
/// <param name="testValue"></param>
/// <returns></returns>
0 references
private string Test(int testValue)
19
{
return testValue.ToString();
}
}

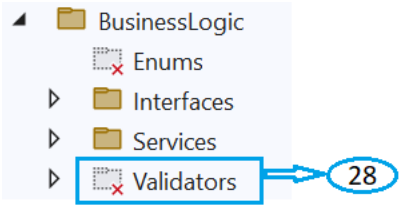







```
/// <summary>
/// Register all Acro Trac DI related module.
/// </summary>
0 references
public static class AcroTracDIRegister
{
    /// <summary>
    /// Register all DI related to acro trac module.
    /// </summary>
    /// <param name="services"></param>
    1 reference
    public static void RegisterAcroTracDI(this IServiceCollection services)
    {
        // Method intentionally left empty to add the DI here.
        services.AddTransient<ISampleService, SampleService>();
    }
}
```






```
public class TestService: IDisposable
{
    private readonly IDisposable _objDispose;
    1 reference
    public TestService()
    {
     // Initialize here
    }
    1 reference
    public void Dispose()
    {
        _objDispose.Dispose();
    }
    0 references
    public void Test1()
    {
        using (var testService = new TestService())
        {
            //Any business logic code here
            testService.Dispose();
        }
    }
}
```

31

31

```
public static class AbcLogMetadata
{
    /// <summary>
    /// Define test log model configuration
    /// </summary>
    /// <param name="modelBuilder"></param>
    0 references
    public static void TestConfigure(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<AccessLog>(entity =>
        {
            entity.HasIndex(x => x.Ukey).IsUnique();
            entity.Property(x => x.DeviceType).HasColumnType("nvarchar(20)").IsRequired(true);
            entity.Property(x => x.LogInTime).HasColumnType("datetime2").IsRequired(false);

        }).HasDefaultSchema("log");
    }
}
```

38

34

33

37