# My Dependency injection:

A dependency is an object that another object depends on. Examine the following MyService class with a WriteMessage method that other classes depend on:

public class IndexModel :PageModel

{

private readonly MyService \_service = new MyService ();

public void OnGet()

{

\_service.WriteMessage("IndexModel.OnGet");

}

}

public class MyService

{

public void WriteMessage(string message)

{

Console.WriteLine($"MyService.WriteMessage called. Message: {message}");

}

}

The class creates and directly depends on the MyService class. Code dependencies, such as in the previous example, are problematic and should be avoided for the following reasons:

1. To replace MyService with a different implementation, the IndexModel class must be modified.
2. If MyService has dependencies, they must also be configured by the IndexModel class. In a large project with multiple classes depending on MyService, the configuration code becomes scattered across the app.This implementation is difficult to unit test.

**Dependency injection addresses these problems through:**

1. The use of an interface or base class to abstract the dependency implementation.
2. Registration of the dependency in a service container. ASP.NET Core provides a built-in service container, IServiceProvider. Services are typically registered in the app's Program.cs file.
3. Injection of the service into the constructor of the class where it's used. The framework takes on the responsibility of creating an instance of the dependency and disposing of it when it's no longer needed.

**HttpContext:**

HttpContext object holds information about the current HTTP request.

It can hold information like, Request, Response, Server, Session, Item, Cache, User's information like authentication and authorization and much more.

public interface text

{

void print();

}

class format : text

{

public void print()

{

Console.WriteLine(" here is text format");

}

}

// constructor injection

public class constructorinjection

{

private text \_text;

public constructorinjection(text t1)

{

this.\_text = t1;

}

public void print()

{

\_text.print();

}

}

class constructor

{

static void Main(string[] args)

{

constructorinjection cs = new constructorinjection(new format());

cs.print();

Console.ReadKey();

}

}

public class logService : IMyService

{

private readonly ILogger<MyService2> \_logger;

public MyDependency2(ILogger<MyService2> logger)

{

\_logger = logger;

}

public void WriteMessage(string message)

{

\_logger.LogInformation( $"MyService2.WriteMessage Message: {message}");

}

}

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();

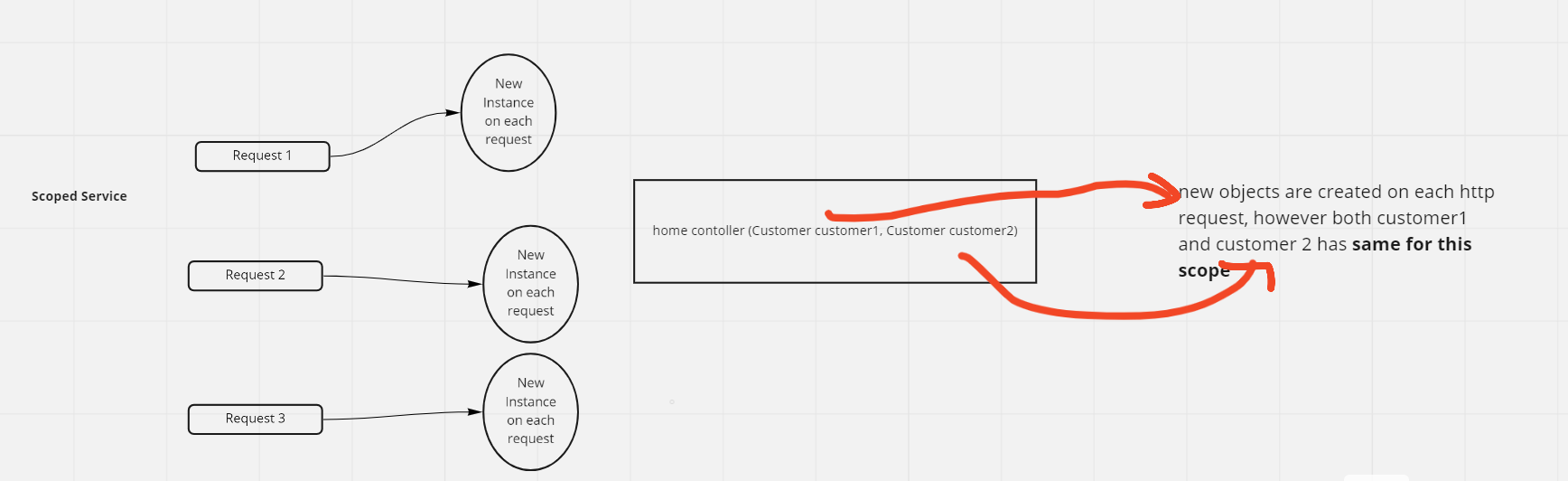
builder.Services.AddScoped<IMyService, MyService>();

var app = builder.Build();

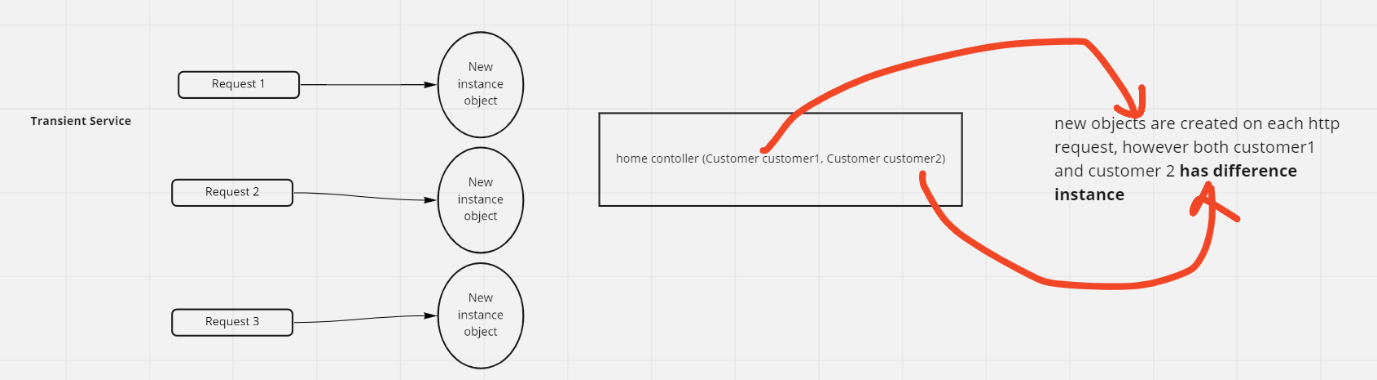
**Three types of lifetime and registration options:**

* **Scoped:**

services.AddScoped<ICronJobService,CronJobService>();



* **Transient:**

services.AddTransient<ICronJobService,CronJobService>();

* **Singleton:**

services.AddSingleton<ILoggingService, LoggingService>();

Only one service instance was created throughout the lifetime.

Reused the same instance in future, wherever the service is required

Since it's a single lifetime service creation, memory leaks in these services will build up over time.

Also, it has memory efficient as they are created once reused everywhere.

**When to use which Service**

Singleton approach => We can use this for logging service, feature flag(to on and off module while deployment), and email service

Only one service instance was created throughout the lifetime.

Reused the same instance in future, wherever the service is required

Since it's a single lifetime service creation, memory leaks in these services will build up over time.

Also, it has memory efficient as they are created once reused everywhere.

Scoped approach => This is a better option when you want to maintain a state within a request.

Transient approach => Use this approach for the lightweight service with little or no state.

Public interface ITransientService {

Guid GetOperationID();

}

Public interface IScopedService {

Guid GetOperationID();

}

Public interface ISingletonService {

Guid GetOperationID();

}

HomeController(ILogger<HomeController> logger,

ITransientService transientService1,

ITransientService transientService2,

IScopedService scopedService1,

IScopedService scopedService2,

ISingletonService singletonService1,

ISingletonService singletonService2)

{

\_logger = logger;

\_transientService1 = transientService1;

\_transientService2 = transientService2;

\_scopedService1 = scopedService1;

\_scopedService2 = scopedService2;

\_singletonService1 = singletonService1;

\_singletonService2 = singletonService2;

}

public class OperationService: ITransientService, IScopedService, ISingletonService {

Guid id;

Public OperationService() {

id = Guid.NewGuid();

}

public Guid GetOperationID() {

return id;

}

}

**Exercise:**

services.AddTransient<ITransientService, OperationService>();

services.AddScoped<IScopedService, OperationService>();

services.AddSingleton<ISingletonService, OperationService>();

Create Asp.net core web api project

Create Token Class that contains Email and company.

Add it as scoped service

Add middleware that adds serialized token Header (Where should it be in the pipline?)

httpContext.Request.Headers.Add("Token", JsonConvert.SerializeObject(t));

Build a middleware that Fills the token from header Data;

Inject it to the controller’s Constructor.