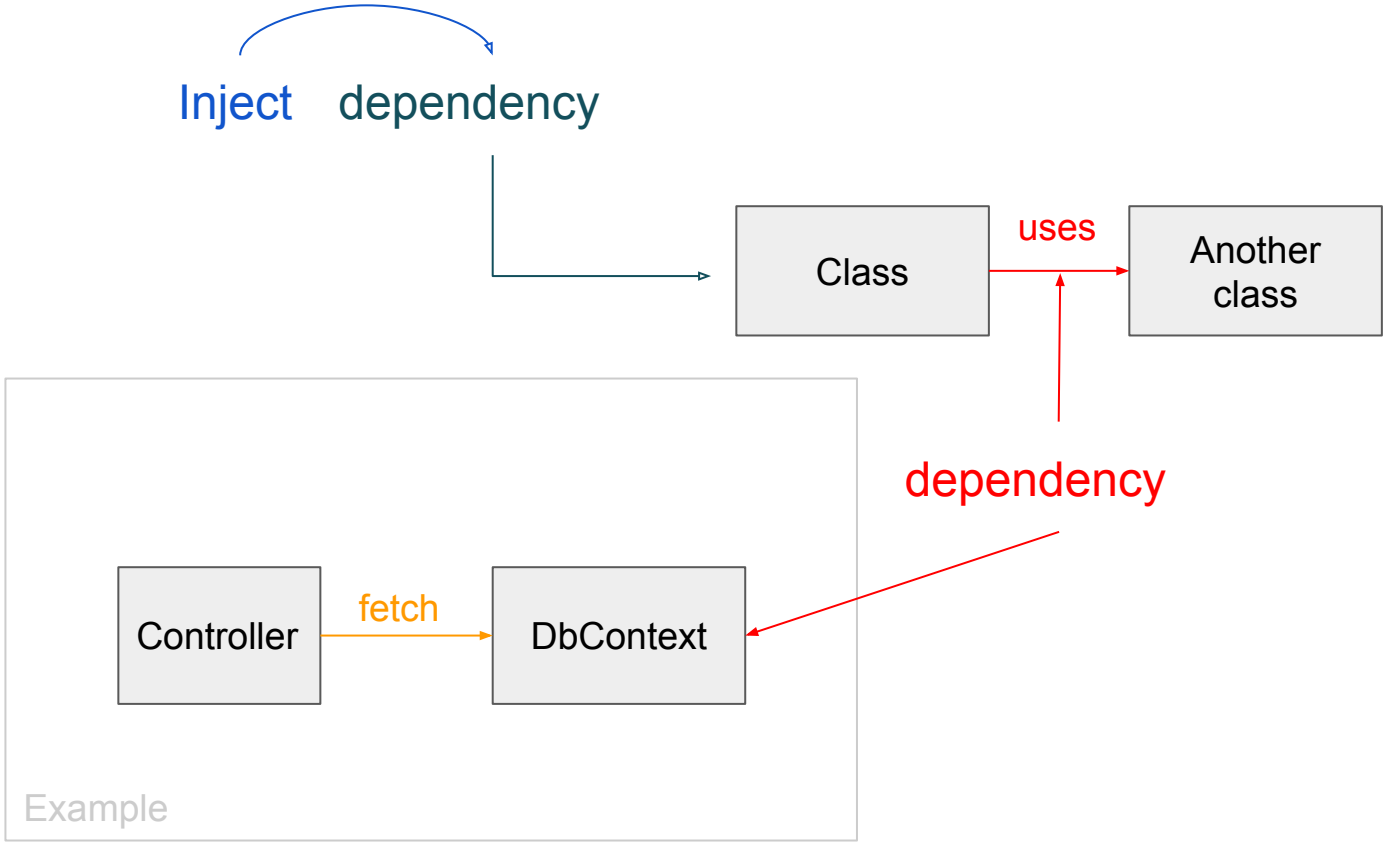
Dependency injection

.net core

Dependency injection



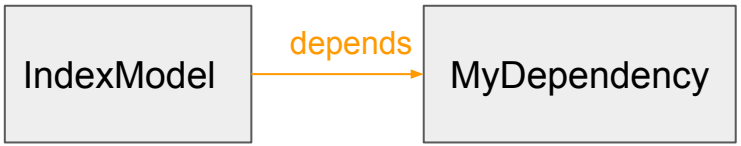
```
public class MyDependency
{
    public MyDependency()
    {
    }

    public Task WriteMessage(string message)
    {
        Console.WriteLine(
            $"MyDependency.WriteMessage called. Message: {message}");

        return Task.FromResult(0);
    }
}
```

```
public class IndexModel : PageModel
{
    MyDependency _dependency = new MyDependency();

    public async Task OnGetAsync()
    {
        await _dependency.WriteMessage(
            "IndexModel.OnGetAsync created this message.");
    }
}
```



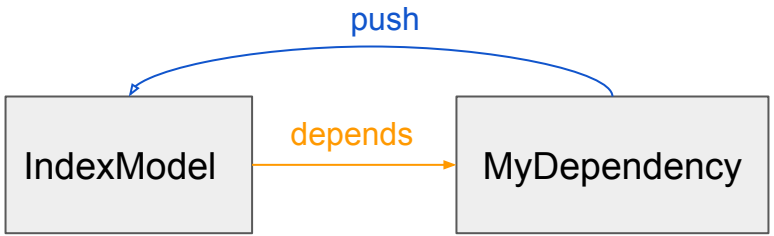
Example

Dependency

A *dependency* is any object that another object requires

- To replace `MyDependency` with a different implementation, the class must be modified.
- If `MyDependency` has dependencies, they must be configured by the class. In a large project with multiple classes depending on `MyDependency`, the configuration code becomes scattered across the app.
- This implementation is difficult to unit test. The app should use a mock or stub `MyDependency` class, which isn't possible with this approach.

Injecting dependencies

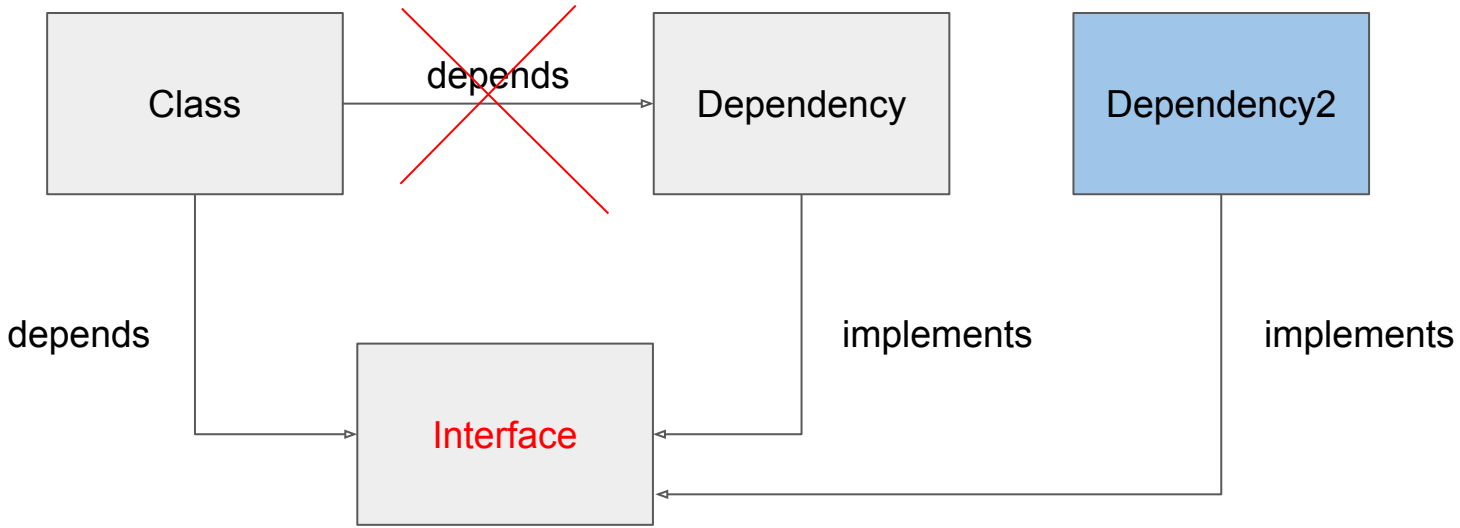


Don’t create dependency inside a class

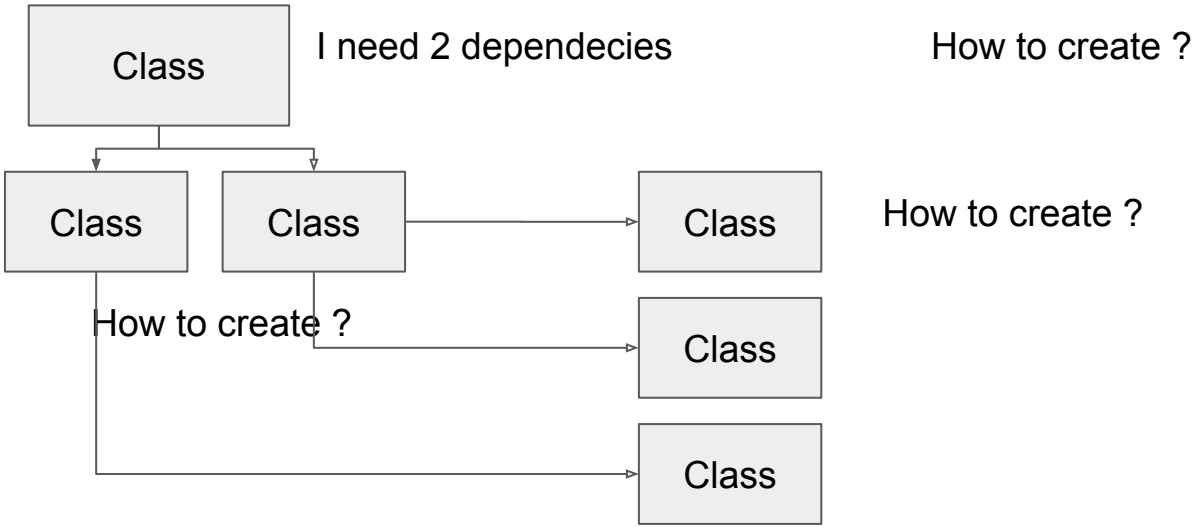
Pass as parameter to constructor

Dependency inversion principle

SOLI**D**



Another problem

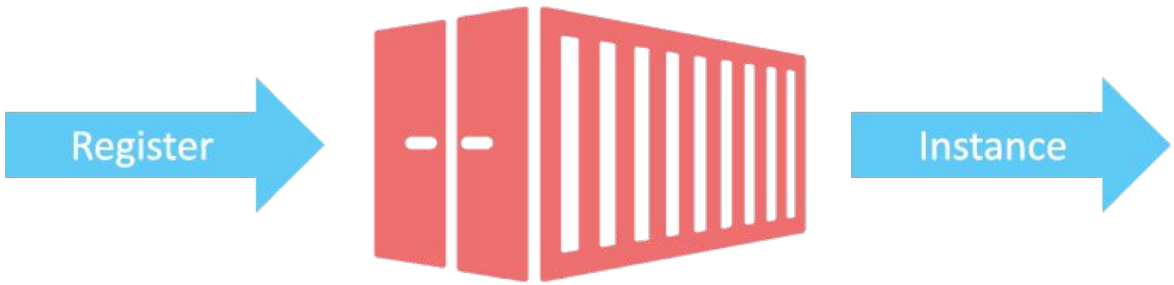


Dependency injection container (Service locator)

“IDependency1” => “Implementation1”

“IDependency2” => “Implementation2”

“IDependency3” => “Implementation3”




ENOUGH BLAH BLAH
SHOW ME SOME CODE!
makeameme.org

```
public interface IMyDependency
{
    Task WriteMessage(string message);
}
```

```
public class MyDependency : IMyDependency
{
    private readonly ILogger<MyDependency> _logger;

    public MyDependency(ILogger<MyDependency> logger)
    {
        _logger = logger;
    }

    public Task WriteMessage(string message)
    {
        _logger.LogInformation(
            "MyDependency.WriteMessage called. Message: {MESSAGE}",
            message);

        return Task.FromResult(0);
    }
}
```

`IMyDependency` and `ILogger<TCategoryName>` must be registered in the service container

```
services.AddSingleton(typeof(ILogger<T>), typeof(Logger<T>));
```

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();

    services.AddScoped<IMyDependency, MyDependency>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new Operation(Guid.Empty));

    // OperationService depends on each of the other Operation types.
    services.AddTransient<OperationService, OperationService>();
}
```

If the service's constructor requires a built in type, such as a `string`, the type can be injected by using configuration or the options pattern:

```
public class MyDependency : IMyDependency
{
    public MyDependency(IConfiguration config)
    {
        var myStringValue = config["MyStringKey"];

        // Use myStringValue
    }

    ...
}
```

Register additional services with extension methods

```
public void ConfigureServices(IServiceCollection services)
{
    ...

    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

    ...
}
```

Service lifetimes

Transient

Transient lifetime services (AddTransient) are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services.

Scoped

Scoped lifetime services (AddScoped) are created once per client request (connection).

Singleton

Singleton lifetime services (AddSingleton) are created the first time they're requested (or when `Startup.ConfigureServices` is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

Service registration methods

Method	Automatic object disposal	Multiple implementations	Pass args

`Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>()`
Example:
`services.AddSingleton<IMyDep, MyDep>();`
	Yes	Yes	No

`Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION})`
Examples:
`services.AddSingleton<IMyDep>(sp => new MyDep());`
`services.AddSingleton<IMyDep>(sp => new MyDep("A string!"));`
	Yes	Yes	Yes

`Add{LIFETIME}<{IMPLEMENTATION}>()`
Example:
`services.AddSingleton<MyDep>();`
	Yes	No	No

`AddSingleton<{SERVICE}>(new {IMPLEMENTATION})`
Examples:
`services.AddSingleton<IMyDep>(new MyDep());`
`services.AddSingleton<IMyDep>(new MyDep("A string!"));`
	No	Yes	Yes

`AddSingleton(new {IMPLEMENTATION})`
Examples:
`services.AddSingleton(new MyDep());`
`services.AddSingleton(new MyDep("A string!"));`
	No	No	Yes

Entity Framework contexts

Entity Framework contexts are usually added to the service container using the scoped lifetime because web app database operations are normally scoped to the client request. The default lifetime is scoped if a lifetime isn't specified by an AddDbContext<TContext> overload when registering the database context. Services of a given lifetime shouldn't use a database context with a shorter lifetime than the service.

Design services for dependency injection

Best practices are to:

- Design services to use dependency injection to obtain their dependencies.
- Avoid stateful, static method calls.
- Avoid direct instantiation of dependent classes within services. Direct instantiation couples the code to a particular implementation.
- Make app classes small, well-factored, and easily tested.

If a class seems to have too many injected dependencies, this is generally a sign that the class has too many responsibilities and is violating the Single Responsibility Principle (SRP). Attempt to refactor the class by moving some of its responsibilities into a new class. Keep in mind that Razor Pages page model classes and MVC controller classes should focus on UI concerns. Business rules and data access implementation details should be kept in classes appropriate to these separate concerns.