# Dependency Injection

## C# Web 2

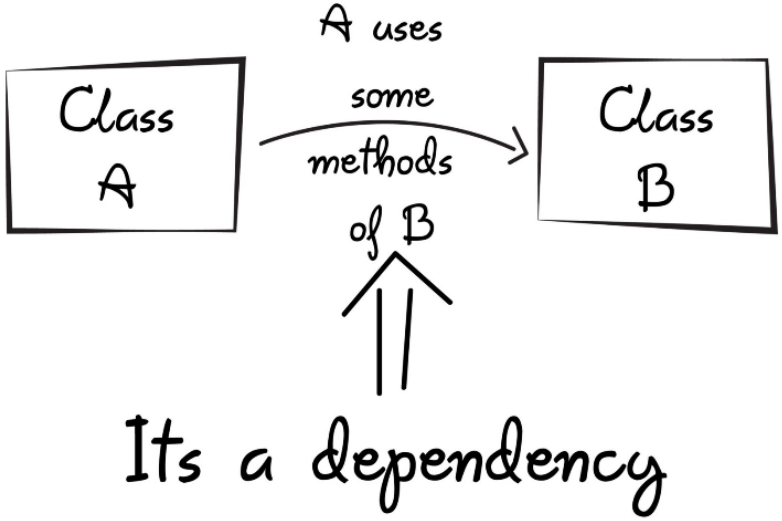
DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

# Dependencies

Als een klasse A in zijn code gebruik maakt van een bepaalde functionaliteit (property, methode) van een andere klasse B, dan wordt er gezegd dat **klasse A afhankelijk** is van **klasse B**.

Of in het Engels: **class A has a dependency on class B**



# Dependency Injection

Vooraleer klasse A gebruik kan maken van bv. een methode van klasse B moet er eerst een object (instantie) van klasse B gemaakt worden.

Klassiek kan je dit doen door in de code van klasse A de instantie aan te maken:

```
public class B
{
    public int CalculateSomething()
    {
        return 1 + 1;
    }
}


public class A
{
    public A()
    {
        var b = new B();
        int result = b.CalculateSomething();
    }
}
```

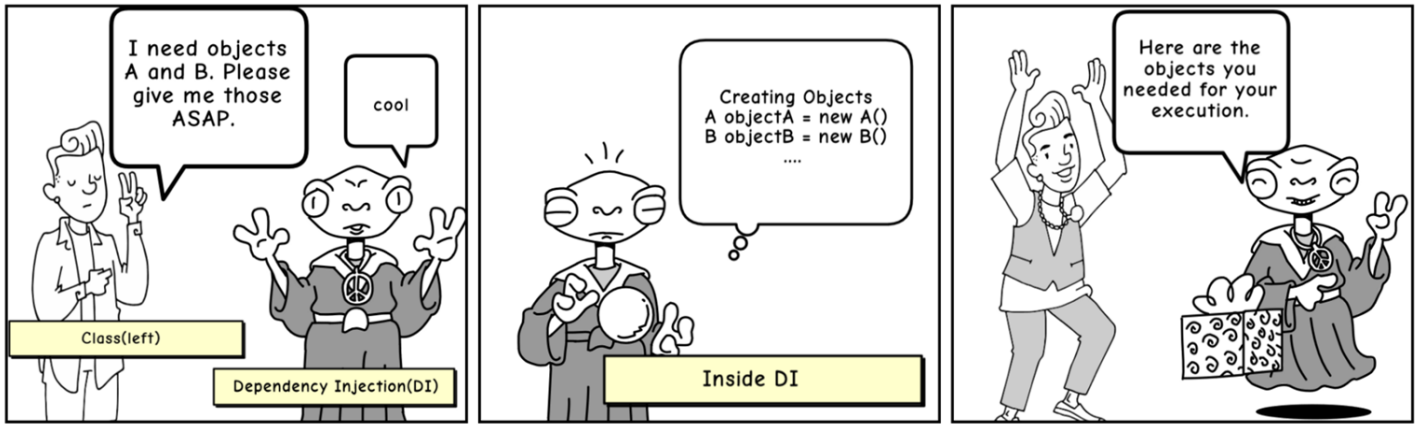
# Dependency Injection

Een alternatief is dat klasse A het klasse B object niet zelf aanmaakt, maar het laat injecteren via, bijvoorbeeld, de constructor. De dependency (klasse B) wordt **geïnjecteerd**:

```
public class A
{
    public A(B b)
    {
        int result = b.CalculateSomething();
    }
}
```

# Dependency Injection

- Waar wordt de instantie van klasse B dan aangemaakt?
  - Bij dependency injection wordt de verantwoordelijkheid voor het aanmaken van een object op 1 centrale plaats gelegd: een **“container”**.
    - In de container wordt beschreven welke klassen allemaal deelnemen aan het dependency injection systeem en wat de levensduur is van aangemaakte objecten van die klassen.
  - Als een klasse een depencency nodig heeft, dan wordt deze dependency door de container aangemaakt en geïnjecteerd in de klasse.



# Dependency Injection

- ASP.NET core heeft een ingebakken dependency injection systeem
  - Dependencies worden hier ***Services*** genoemd
  - In **Program.cs** laden we de services in de builder.Services property. Deze property is van het type ***IServiceCollection***. Dit is de container waaraan we dependencies kunnen toevoegen.
  - Waarin injecteren?
    - Controllers
    - Nieuwe Services

# Dependency Injection

- Voorbeeld:
  - *ProductRepository* class gebruiken die als dependency geïnjecteerd zal worden in een controller.

  - Normaal zouden we in onze controller gebruik moeten maken van ProcuctRepository repo= new ProductRepository() om het object aan te maken en te kunnen gebruiken.

  - Door gebruik te maken van Dependency Injection wordt het object echter aan gemaakt door de container en krijgt het object een bepaalde **levensduur**.

# Dependency Injection - Lifetime

Stel je hebt een service nodig van de container.
Afhankelijk van de levensduur van een service, gaat de container dan beslissen of het nodig is om een nieuw object aan te maken of niet.

Er zijn **3 mogelijke levensduren**:

- **Transient**
  - De container maakt elke keer een nieuw object aan als er een instantie van een service (dependency) gevraagd wordt.
- **Scoped**
  - Bij een scoped service (dependency) wordt hetzelfde object steeds teruggegeven zolang dit gevraagd wordt binnen dezelfde http request. Als er een nieuwe http request (een nieuwe scope) binnenkomt, dan wordt er weer een nieuwe instantie gemaakt die dan steeds wordt teruggegeven tijdens het verwerken van die http request.
- **Singleton**
  - De eerste keer dat een singleton service wordt gevraagd, wordt er een nieuw object gemaakt. Alle volgende keren wordt ditzelfde object teruggegeven.

# Dependency Injection - Abstraction

Voordelen:

- Code is makkelijker leesbaar.
  Classes die dependency gebruiken moeten zelf geen objecten meer aanmaken.
- Makkelijk polymorfisme via interfaces.

De container bepaalt welk concreet object (die de interface implementeert) aangemaakt wordt.

```
public class Golfer
{
    public Golfer(IClub
club)
    {
        club.HitBall();
    }
}
```

```
public interface IClub
{
    void HitBall();
}
```

```
public class IronClub :
IClub
{
    public void HitBall()
    {
       //hit the ball
    }
}
```

# Dependency Injection - Abstraction

Afhankelijk zijn van een interface en niet van een concreet type, brengt de volgende extra voordelen:

- Klassen zijn meer losgekoppeld van elkaar **(loose coupling**). Hierdoor wordt de code beter uitbreidbaar en beter onderhoudbaar
- Het wordt gemakkelijker om automatische testen (**unit testen**) te schrijven die de klassen testen ⇒ **mocking**

ASP.Net Core MVC - MVCDependencyInjection

# Visual Studio

- Visual Studio – **MVCDependencyInjection**
  - ASP .Net Core Web application
  - MVC
- Visual Studio - Project
  - Models
    - Add Class – Product
  - Services
    - Add Interface IProductRepository
    - Add Class - ProductRepository

**Entities – Add class Product.cs**

```
namespace MVCDependencyInjection.Models
{
    public class Product
    {
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

**Services – Add Class IRepository**

```
namespace MVCDependencyInjection.Services
{
    public interface IProductRepository
    {
        IEnumerable<Product> Products { get; }
        Product this[string name] { get; }
        void Add(Product product);
        void Delete(Product product);
    }
}
```

**Services – Add class – ProductRepository**

```
namespace MVCDependencyInjection.Services
{
    public class ProductRepository : IProductRepository
    {
        private Dictionary<string, Product> products;
        public ProductRepository()
        {
                                Product football = new Product();
                                football.Name = "football";
                                football.Price = 10m;

                                Product tennisball = new Product();
                                tennisball.Name = “tennisball";
                                tennisball.Price = 3m;

                                products = new Dictionary<string, Product>();
                                products.Add(football.Name, football);
                                products.Add(tennisball.Name, tennisball);
        }
        public IEnumerable<Product> Products => products.Values;
        public Product this[string name] => products[name];
        public void Add(Product product) => products[product.Name] = product;
        public void Delete(Product product) => products.Remove(product.Name);
    }
}
```

ASP.Net Core MVC – Startup.cs – Dependency Injection - Repository

# ADDTRANSIENT

**Dependency Injection – Program.cs**

```
builder.Services.AddControllersWithViews();
builder.Services.AddTransient<IProductRepository,
    ProductRepository>();
```

**Views/Home/Index.cshtml**

```
@{
    int productCount = (int)ViewBag.ProductCount;
    ViewData["Title"] = "Home Page";
}
<div class="text-center">
    <h1 class="display-4">Products</h1>
    <p>
        @productCount unique products in our repository
    </p>
</div>
```

**Controllers/HomeController.cs**

```
public class HomeController : Controller
{
    private IProductRepository _repo;
    public HomeController(IProductRepository repo)
    {
        _repo = repo;
    }
    public IActionResult Index()
    {
        ViewBag.ProductCount = _repo.Products.Count();
        return View();
    }
}
```

**Views**

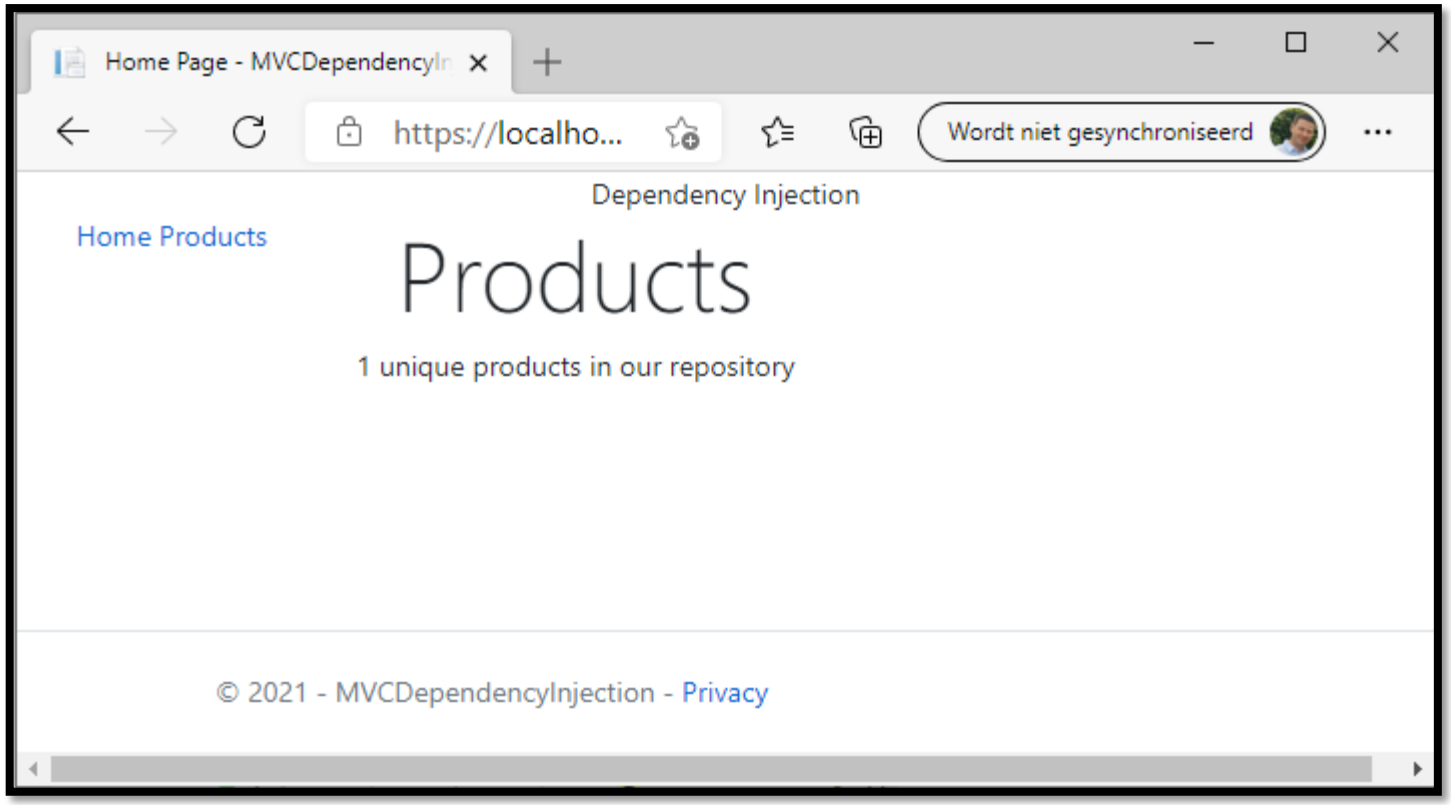
- **Create Folder Products**
  - **Create View – Create template – Create.cshtml**
  - **List View – List template – Index.cshtml**

**ProductsController**

```
public class ProductsController : Controller
{
    IProductRepository _repo;
    public ProductsController(IProductRepository repo)
    {
        _repo = repo;
    }
    public IActionResult Index()
    {
        return View(_repo.Products);
    }
    public IActionResult Create()
    {
        Product p = new Product();
        return View(p);
    }
    [HttpPost]
    public IActionResult Create(Product p)
    {
        _repo.Add(p);
        return RedirectToAction("Index", "Products");
    }
}
```

Views/Shared/_Layout.cshtml

```
<a asp-controller="Home" asp-action="Index">Home</a>
<a asp-controller="Products" asp-action="Index">Products</a>
```



# Service type: AddTransient

Vermits we gebruik maken van AddTransient als service type in onze service gaat hij iedere keer opnieuw het object aanmaken.
Onze repository blijft dus 1 product bevatten bij het herladen.

ASP.Net Core MVC – Dependency Injection - Repository

# ADDSCOPED

**Dependency Injection – Program.cs**

```
builder.Services.AddControllersWithViews();
builder.Services.AddScoped<IProductRepository, ProductRepository>();
```

# Service type: AddScoped

Vermits we gebruik maken van AddScoped als service type in onze service gaat hij bij elke nieuwe http request opnieuw het object aanmaken.
Onze repository blijft dus 1 product bevatten bij het herladen.

**Dependency Injection – Program.cs**

```
builder.Services.AddControllersWithViews();
builder.Services.AddSingleton<IProductRepository,
    ProductRepository>();
```

# Service type: AddSingleton

Door gebruik te maken van AddSingleton als service type in onze service gaat hij het object slechts 1 keer aanmaken en daarna telkens hetzelfde object gebruiken. Op deze manier kunnen we producten toevoegen en verwijderen uit onze repository gedurende het gebruik van onze toepassing.
Eén en hetzelfde repository object wordt nu dus gemanipuleerd bij het uitvoeren van de acties in onze ProductsController.

ASP.Net Core MVC – Dependency Injection
Oefening 02

# Dependency Injection